AFRL-RH-BR-TR-2010-0001

Optimizer User's Manual

Albert Bailey
Northrop Grumman Information Technology
4241 Woodcock Drive, Ste. B-100
San Antonio, TX 78228

Human Effectiveness Directorate
Directed Energy Bioeffects Division
Optical Radiation Branch
2624 Louis Bauer Drive
Brooks City-Base, TX 78235-5128

December 2009

Interim Report for November 2008 to January 2009

Air Force Research Laboratory
711 Human Performance Wing
Human Effectiveness Directorate
Directed Energy Bioeffects
Optical Radiation Branch Brooks City-Base, TX 78235

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 311th Public Affairs Office at Brooks City Base, TX and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RH-BR-TR-2010-0001 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

_____//Signed//_____
Samuel Y. O, Lt, USAF
Work Unit Manager
711 HPW/ RHDO

_____//Signed//_____
GARRETT D. POLHAMUS, Ph.D.
Chief, Directed Energy Bioeffects Division
Human Effectiveness Directorate
711 Human Performance Wing
Air Force Research Laboratory

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
10-12-2009	Interim Technical Report	November 2008-January 2009

4. TITLE AND SUBTITLE

Optimizer User's Manual

5a. CONTRACT NUMBER
FA8650-08-D-6930

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
0602202F

6. AUTHOR(S)

Albert Bailey

5d. PROJECT NUMBER
5020

5e. TASK NUMBER
D2

5f. WORK UNIT NUMBER
05

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
711 Human Performance Wing
Human Effectiveness Directorate
Directed Energy Bioeffects
Optical Radiation Branch
Brooks City-Base, TX 78235-5214

Northrop Grumman -IT
4241 Woodcock Dr., Ste B-100
San Antonio, TX 78228

8. PERFORMING ORGANIZATION REPORT

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
711 Human Performance Wing
Human Effectiveness Directorate
Directed Energy Bioeffects
Optical Radiation Branch
Brooks City-Base, TX 78235-5214

10. SPONSOR/MONITOR'S ACRONYM(S)
711 HPW/RHDO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-RH-BR-TR-2010-0001

12. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution A: Approved for public release; distribution unlimited. Approval given by local Public Affairs Office #10-032, 8 February 2010, Brooks City-Base, Texas 78235.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This manual documents the RHDO optimizer code. This code is designed to optimize inputs to client programs to produce desired outputs, using genetic algorithms to guide the search.
The Argonne National Laboratory PGAPack library is used to determine the optimization search values. It works on cluster computers as well as stand-alone PCs. The code is written in C++ to run under Microsoft Windows or Linux with no source code modifications.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Samuel Y. O
U	U	U	SAR	37	**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. 239.18

This Page Intentionally Left Blank

TABLE OF CONTENTS

LIST OF FIGURES

This Page Intentionally Left Blank

ABSTRACT

This manual documents the RHDO optimizer code. This code is designed to optimize inputs to client programs to produce desired outputs, using genetic algorithms to guide the search.

The Argonne National Laboratory PGAPack library is used to determine the optimization search values. It works on cluster computers as well as stand-alone PCs. The code is written in C++ to run under Microsoft Windows or Linux with no source code modifications.

1. INTRODUCTION

Computer simulations allow predictions of effects produced as a result of certain input conditions. Frequently, the purpose is to determine what input conditions are required to produce a desired effect. This determination can only be achieved by running the simulation multiple times with different input conditions, iteratively improving the guesses for the input conditions until the desired effect is achieved. This iterative process is frequently accomplished manually, but this requires significant human time and effort. The optimization process is frequently better accomplished by using a computer program to control and execute the simulation iterations. This document describes a code produced for this purpose.

This code was constructed to run other programs, changing the input files and examining the output files to optimize some desired condition. The optimizer code is designed to be independent of the code whose inputs are being optimized: it does not need to be linked to the other program and few if any changes to the program should be required.

The program operates by creating input files for the client program, executing that program via a system command, then reading the resulting output files to determine how close the outputs are to the desired values. On a machine or cluster with multiple processors, the optimizer can execute several trials at once, speeding the optimization process. The code works under Microsoft Windows or Linux without modification of the source code.

The code has been initially written to support optimizations using the BTEC code.[1] However, it has been made as general as possible to support use with other programs as well. Doing so would require some new code to be written, since each program has its own way of reading input files and writing output files. The portions that would require alteration have been localized to ease the required modifications.

The core of the optimizer is the Argonne National Laboratories parallel genetic algorithm PGAPack library. The library has been incorporated largely unchanged. (Minor changes were made to convert the library from C to C++.) The library is well documented in the Argonne National Laboratory report ANL-95/18. The original code and documentation are available on the Web at
http://www-fp.mcs.anl.gov/CCST/research/reports_pre1998/comp_bio/stalk/pgapack.html .

2. GENETIC ALGORITHMS

Genetic algorithms are an optimization technique inspired by evolution. They operate by attempting to "breed" a solution. An initial population of trial input values is generated randomly. These trial solutions are tested and the resulting output values computed.

The solutions that are farthest from the desired result are "killed" and the remaining solutions retained. Additional new trial solutions are generated by mutation and cross-breeding of the retained solutions. The new trial solutions are tested and their deviation from the optimal is also recorded. The population of solutions is then culled again, retaining the best sets of input values. The procedure is repeated over and over again. Eventually the surviving trial solutions should be good approximations to the optimum.

Genetic algorithms have the advantage that they are very robust and trustworthy. They are able to find the best solution even in the presence of false minima. They do not require that the solution space be well behaved. Incorporating constraints is not a problem. Trial solutions that are not within constraints are "killed" and are not used in generating new trial solutions.

In general, any kind of data (integer, boolean, float) can be used as a variable for optimization, as long as it can be represented as a bit sequence. The genetic algorithms themselves work without any reference to the type of data under consideration. The only point where the nature of the data becomes significant is in the mapping from the bit sequence to a numerical value. For example, if a variable is optimized in the region between 0 and 1 using an 8 bit sequence, the optimization will be a choice between 256 (2^8) equally spaced values between 0 and 1, separated by 0.0039 (1/256). Any number of optimization variables can be used, combining their bit stream into a single "chromosome" representing the entire description of the solution (Figure 1).

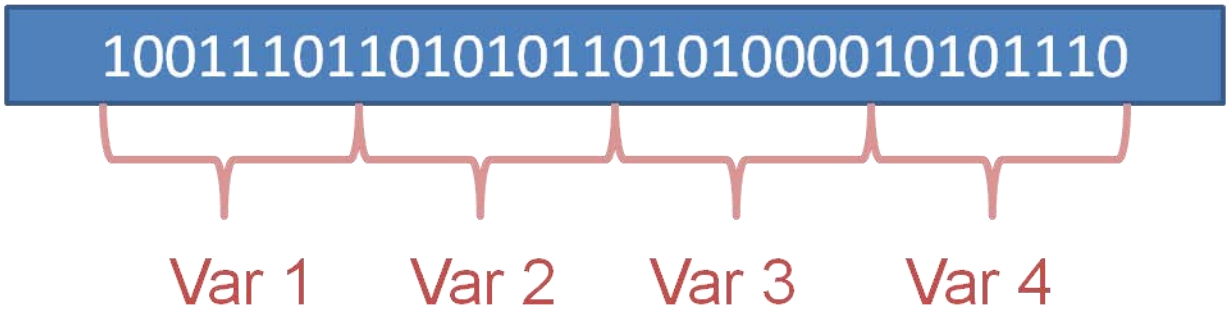

Figure 1. The variable values to be optimized are stored as a bit sequence

The two techniques used to produce new trial solutions are mutation and crossover (Figure 2). Mutation takes an existing trial solution and flips one or more of the bits of the solution. In crossover, a new trial solution is constructed by combining portions of two existing trial solutions.

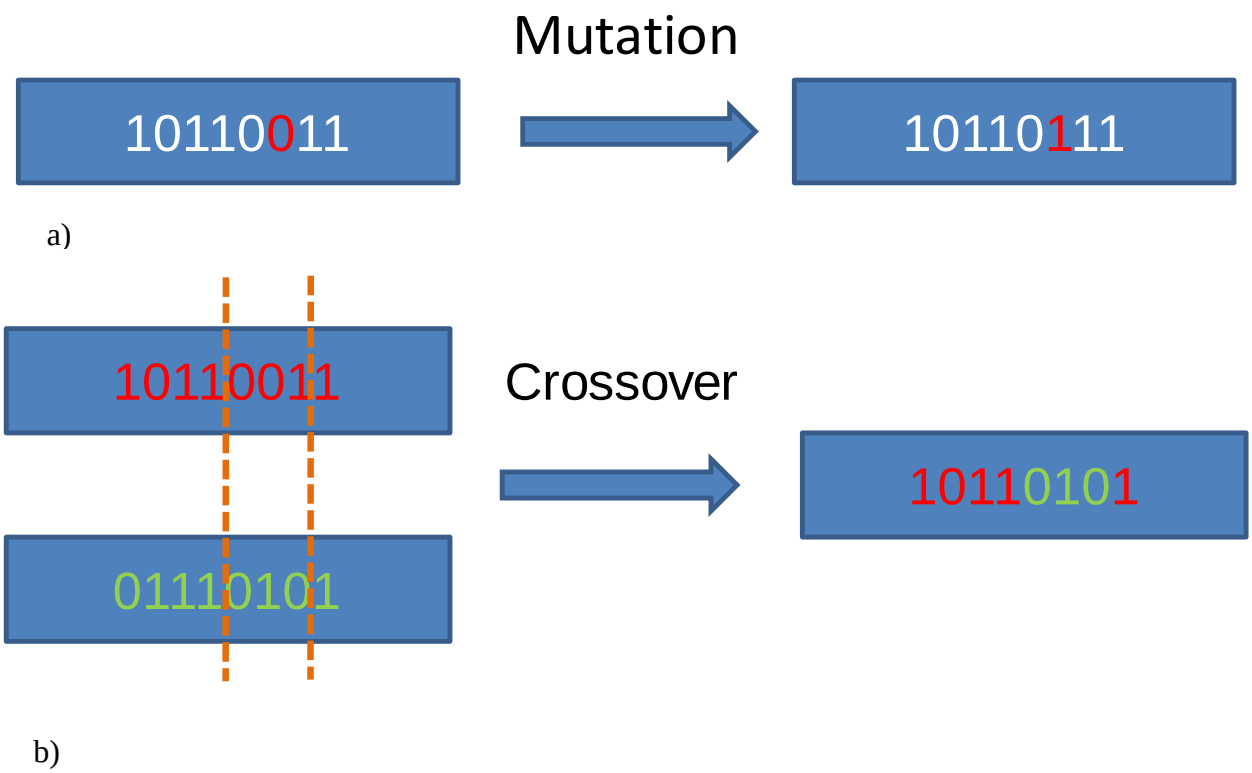

Figure 2. New trial solutions are created by mutation (a) or crossover (b)

3. PGAPACK UTILIZATION

The Optimizer reads in an input file designating the variables to be optimized. This is discussed in detail in a later section. It then calls PGAPack to optimize the client code. PGAPack first constructs a set of trial solutions. The `evaluateCase` function is called by PGAPack for each new trial solution that is constructed. This function creates one or more new input files with the solution variables inserted for the dummy variables. The routine then uses a `system` call to run the client program, using these new input files. The output files are then read and the quality of the solution is judged, and returned to the PGAPack library as the return value. A value of zero represents the idealized goal, any actual value (even the optimum attainable) being higher. This entire process is enclosed in a `try ... catch` block, so that even if the procedure fails for some reason, the optimizer code does not crash: it simply returns a set large value (`BADFAIL`) as the result. It is possible for the solution variables to be so inappropriate that the client program crashes. The routine checks the return value of the `system` call, and throws an exception if zero was not returned. This is treated as any other failure in the `try` block, and the `BADFAIL` value is returned by `evaluateCase`.

The PGAPack routines have been set in the namespace `PGAPack`. The other routines and variables, except for the main program, are in the namespace `Optimizer`. There is one new function, `MyPGARun`, not part of PGAPack but based on routines in PGAPack, which uses the PGAPack namespace.

On a cluster or multi-core machine, multiple cases can be evaluated simultaneously, speeding the execution of the optimization (Figure 3). EvaluateCase creates a separate subdirectory for the input and output files for each process, removing this subdirectory before returning.

To test the quality of a solution, evaluateCase calls the getBadness method of one or more Condition objects. The abstract virtual Condition base class is general, making no assumptions about the client program being run or the types of output files being created. In order to optimize a particular program, one or more subclasses must be created.

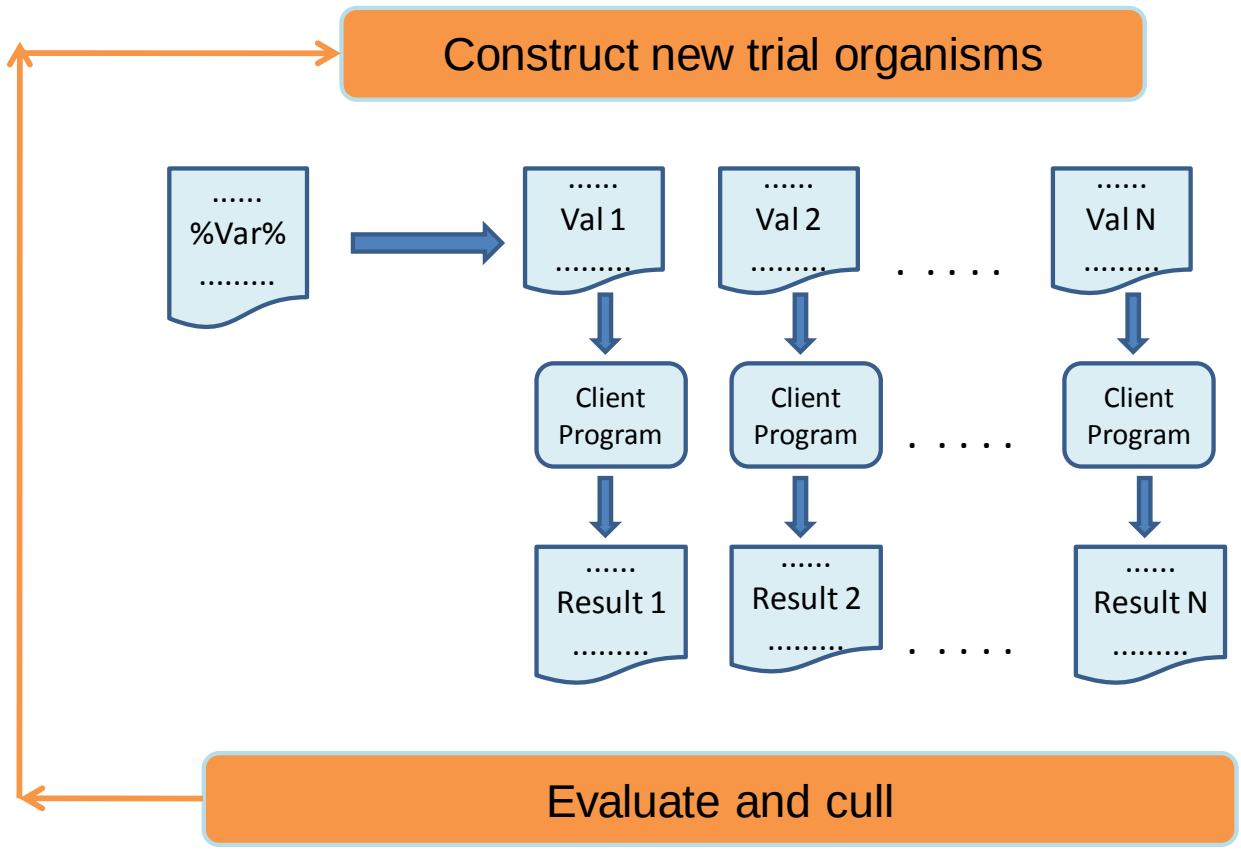

Figure 3. Multiple trial solutions can be evaluated in parallel if multiple processors are available

4. PROGRAM STRUCTURE

The Optimizer code consists of the following sections:

`Optimizer.cpp` – This is the primary core of the code. It contains the main routine which sets up and controls the optimization. In addition to the main routine, this file also holds the `evaluateCase` function that is called by PGAPACK for each new trial solution. The `evaluateCase` routine inserts the trial solution variables in the input file for the client code, executes the code, evaluates the outputs, and returns a measure of the quality of the trial solution. This file also includes a number of ancillary routines and output routines (`makeProcName`, `findAndReplace`, `readInputFile`, `flispslash`, `printCurrentState`, `printFinalState`). This file also includes `MyPGARun`, an altered version of the PGAPack driver routine `PGARunGM`.

`Optimizer.h` – This is header file for the optimizer code.

`Condition.h` – This file defines the abstract Condition base class. For each code to be optimized, subclasses must be created which can read the output files of the client program and determine the degree to which the desired condition has been met. This base class has two virtual methods that must be overloaded by subclasses: `isConstraint()` and `getBadness()`.

Two types of conditions can be created: minimizations and constraints. Constraints are conditions that must be met for a solution to be valid. When the method `getBadness()` is called, constraints should return the predefined large number `BADFAIL` if the constraint is not satisfied and zero if it is satisfied. Optimizations should return a positive value that is lower for a better trial solution. Constraints should return a value of `true` for calls to `isConstraint()`, and optimizations should return a value of `false`.

Each condition is assumed to be able to be determined by evaluating a single output file created by the client program. Multiple constraints may evaluate the same output file or different output files, as desired. Only one minimization condition is meaningful, but multiple constraint conditions are allowed. An optimization can have up to 10 constraints.

`SpecificConditions.h, SpecificConditions.cpp` – These files provide the constraints for a given client program. They must define the condition subclasses, overriding the pure virtual `isConstraint()` and `getBadness()` methods of the `Condition` class.
The header file must also define the values `CONSTRAINT0, CONSTRAINT1, ...` , setting these values to the names of the actual constraint subclasses to be used.

ProgramSpecific.cpp – This file contains the functions `getExecutionString`, `getConditionOutputFile`, and `cleanString` which are specific to the client program to be optimized. `getExecutionString` returns the line for the system call to execute the client program, including all arguments. The input file name and output directory are provided as inputs. `getConditionOutputFile` returns the full filepath-filename string needed to open the output file used for a specific condition. The filename and output directory are provided as inputs. `cleanString` provides for stripping extraneous comments from input files. This was needed by the BTEC program which frequently contains percent (%) signs in comments, which confuse variable replacement. (Percent signs MUST be used ONLY with dummy replacement values for this optimization program to work properly.)

PGAPack.h – This header file is included by `Optimizer.cpp` to interface to the PGAPack code. The remainder of the PGAPack code is best managed by compiling it into a library.

mathexpr.h, mathexpr.cpp – These are freeware routines by Yann Ollivier which allow for mathematical expressions to be used as well as individual values for dummy variable replacement. For example, an input file line might have a key-value pair written as

```
power = %Var1 + (2*Var2)%
```

If the trial solution had Var1 equal to 2.2 and Var2 equal to 1.5 then the input file after replacement would have the key-value pair

```
power = 5.2
```

since 2.2+(2*1.5) = 5.2. This code is available online at
http://www.yann-ollivier.org/mathlib/mathexpr.php

KeyValueFileReader.h, KeyValueFileReader.cpp – These routines allow the input to the optimizer to use the same key-value-comment format as is used in other input files for the BTEC code. These routines are taken directly from the BTEC code repository.

helcat_utilty.h – These are some general utility routines taken from the HELCAT code base. They are used for a variety of simple string manipulations.

5. INPUT FILES

The Optimizer code itself has an input file which controls its operations. It, in turn, will write input files for the client program to produce trial solutions. An example input file used to run the BTEC code is shown below in Figure 4. To execute the code from a command line interface, one would simply type:

`Optimizer` *inputfile*

the `argv[1]` value (*inputfile*) being opened as the input file.

```
#Test input file for optimizer
maxIterations          = 5                    # Maximum number of iteration generations
generationSize         = 30                   # Number of individuals in a generation
numReplace             = 10                   # Number to replace each generation
randomNumberSeed       = 3                    # Change seed to check optimization (integer)
numOptVars             = 5                    # Number of variables to be optimized
optVarMin[0]           = 0.0                  # Laser power mininum
optVarMax[0]           = 10.0                 # Laser power maximum
optVarMin[1]           = 1000.0               # Wavelength minimum
optVarMax[1]           = 4000.0               # Wavelength maximum
optVarMin[2]           = 0.001                # Laser on time minimum
optVarMax[2]           = 10.0                 # Laser on time maximum
optVarMin[3]           = 0.2                  # Pulse period as fraction of laser on time
minimum
optVarMax[3]           = 1.0                  # Pulse period as fraction of laser on time
maximum
optVarMin[4]           = 0.0                  # LOg10(Pulse period/Pulse duration) mininum
optVarMax[4]           = 1.0                  # Log10(Pulse period/Pulse duration) maximum
numConditions          = 2                    # Number of conditions for the optimization
condition[0]_goal      = 1.0                  # Maximum damage allowed at any point in
condition[0]_outputFile       = "Sensor-Scal-2.t.last.btlog"       # Damage threshold probe
condition[1]_goal      = 0.0                  # Not used.  Just minimize
condition[1]_outputFile       = "Sensor-Scal-1.t.last.btlog"       # Temperature rise probe
optOutFile             = "Opt1.opo"           # Optimizer output file
numVarInFiles          = 2                    # Number of input files with varying values
varInFile[0]           = "test.config.in"            # Main program input file
varInFile[1]           = "../StandardEmitter/test.emitter"   # Input file to be altered
workPath               = "./OptTemp" # Name of directory for work files (must already exist)
```

Figure 4 Sample input file to the Optimizer code for the client code BTEC

The program works by generating an initial population of trial solutions. The poorer solutions are discarded and new trial solutions are generated by mutation and crossover to replace them. This represents a new generation. The fitness of the new solutions is examined and the process is repeated. `generationSize` is the total number of trial solutions retained. Each generation, `numReplace` trial solutions are discarded and replaced with new trial solutions. The process is allowed until `maxIterations` generations have been created. The program can stop before `maxIterations` generations have been created if the optimization ceases to improve or if all the individuals in a generation become identical. When running on multiprocessors, the optimum size for `numReplace` is one less than the total number of processors used. This allows a separate case to run on each processor without interference. (The root processor does not do evaluations when the number of available processors is greater than two.) `generationsize` should be at least twice `numReplace` to avoid discarding better solutions to replace them with worse ones.

`randomNumberSeed` is an integer variable used to initialize the random number generator, which generates a random set of initial trial solutions, and to produce random mutations and crossovers. Rerunning an optimization with a different `randomNumberSeed` value is a good check as to whether the Optimizer has converged to the correct solution.

`numOptVars` represents the number of variables being independently changed in the optimization. Additional variables may be changed as a result, if they depend on those values. For example, if laser pulse duration and laser pulse repetition rate are independent variables,

laser duty cycle would be set as a dependent variable. The `mathexpr` library provides a simple way to set these dependent variables. For each independent variable, a minimum value `optVarMin` and a maximum value `optVarMax` must be specified. When searching over many orders of magnitude, it can be more advantageous to set the independent variable as the common logarithm of the variable being sought. The program is dimensioned to allow for up to ten independent variables (`Var0, Var1, Var2, ... Var9`). While the program will work even for searches with only a single independent variable, it is not the optimum method for searching under such a situation.

`numConditions` represents the number of conditions for a solution. Exactly one of these should be a minimization criterion; the rest should be constraints. Up to ten conditions (numbered 0 to 9) can be set. Each condition must have value of `CONDITION#` (# = 0 to 9) defined in `SpecificConditions.h` and must be defined to a Condition subclass type defined therein. For each condition there also needs to be an output file that is examined (`condition[#]_outputFile`) to determine how well the trial solution meets the condition, and a goal (`condition[#]_goal`) for the condition. For optimizations, the Condition should strive to reach the goal. For constraints, the goal is a level that either must be met or must not be exceeded.

In order for the differing input variable to be input into the client program, the optimizer will need to rewrite one or more input files read by the client program. `numVarFiles` represents the number of different input files that must be rewritten to specify a trial solution. For each file that is to be rewritten, the name of the input file `varInFile[#]` (# = 0 to 9) should be specified. There is also provision for an overall working directory, `workPath`, to be specified as well.

For example, shown below is an input configuration file (`test.config.in`) for the BTEC code, one of the files designated in the input file to the optimizer itself. This code, as is, is not an acceptable input file for BTEC. In two places, **boldfaced** and in red, the values of two key-value pairs have been replaced with placeholders. The placeholders are designated by being fenced with percent (%) signs. The optimizer code will read in this file and store it in a string. For each evaluation of a trial solution, the placeholder values will be replaced and the resulting string written out to form an input file for BTEC. The first placeholder will be replaced by a value dependent on the second independent variable (Var2) of the optimization. The second placeholder is replaced by the file designated in `varInFile[1]`. The other independent variables not used in this file have placeholders that use them within the file designated by `varInFile[1]`.

```
#KeyValue config file
Dimensions             = 1       # INT: number of dimensions to use (1 | 2)
SimulationType         = "Thermal"       # INT: problem (sim) type
AxialGridType          = 0            # INT: 0 = UNIFORM, 1 = STRETCHED
Nz                     = 600          # INT: number of z grid divisions
zMin                   = 0            # DBL: minimum z coordinate
zMax                   = 0.525             # DBL: maximum z coordinate
zStretchRatio          = 1.05         # DBL: z stretch ratio for
zMinBC                 = 7            # INT: z min boundary condition type
zMaxBC                  = 0           # INT: z min boundary condition type
RadialGridType         = 0            # INT: radial grid type
Nr                     = 200          # INT: number of r grid divisions
rMax                   = 1.0            # DBL: Max r coordinate
rStretchRatio          = 1.02         # DBL: r stretch ratio for non-uniform grid
rMaxBC                 = 0            # INT: r max boundary condition type
TotalSimTime            = %Var2 + 0.1% # DBL: total simulation time
dt                      = 0.001       # DBL: time step for fixed emitter, not for adaptive
dtMax                   = 0.010       # DBL: maximum time step
TissueBaseTemp          = 37          # DBL: tissue baseline temperature
AmbientTemp             = 20          # DBL: ambient temperature
RelHumidity             = .50         # DBL: relative humidity
LogDataFlag             = 1           # INT: log data flag (0 = NO, 1=YES)
LogInterval             = 0           # INT: log interval in time steps
stepsMaxPowerRatio          = 10.0        # DBL: max power ratio
MinPowerRatio          = 0.1         # DBL: min power ratio
ConvergeThresh         = 0.10        # DBL: convergence threshold for search
Sensor[0]              = "../Sensor/test.sensor.1.btec"
Sensor[1]              = "../Sensor/test.sensor.2.btec"
Sensor[2]              = "../Sensor/test.sensor.3.btec"
StandardEmitter[0]     = "%File1%"            # STR: emitter file


          Layer[0]               = "../Layer/layer1.layer"    # STR: layer filename
Layer[1]               = "../Layer/layer2.layer"    # STR: layer filename
Layer[2]               = "../Layer/layer3.layer"    # STR: layer filename
Layer[3]               = "../Layer/layer4.layer"    # STR: layer filename
InitialConditionsFlag  = 1             # INT: initial conditions flag
InitialConditionsFile  = "../InitialConditionsFile/37_20_50_7_0.axialTemp.t.last.btlog" " #
```

Figure 5 Sample input file for BTEC with placeholder values shown in bold red

It is important that the input files not have percent (%) signs in the variable names or values, since this is the mark the code uses to determine placeholders. The code has been written in such a way as to allow for percent signs in comments: when reading in the initial input files, all the comments are stripped using the `cleanString` routine. In adapting the optimizer to work with other client codes, similar measures might be required.

6. FUTURE DIRECTIONS

The Optimizer code has been initially constructed to work with the BTEC code. With minor modifications, it could also be used as an optimizer for other RHDO codes. This would likely require routines to deal with other input and output file formats, such as XML.

The genetic algorithm technique that has been used is very robust, but not very efficient. In the future it might be desirable to incorporate a faster routine to refine the optimum once the genetic algorithm technique has discovered the rough location in parameter space of the minimum. The technique is also not optimal where only one or two variables need to be optimized. In these situations, a bracketing technique might be more appropriate.

References

1. L. Irvin, P.D.S. Maseberg, G. Buffington, C.D. Clark III, R. J. Thomas, M. L. Edwards, J. Stolarski "BTEC Thermal Model", report AFRL-RH-BR-TR-2008-0006, 2008.

Appendix A: Optimizer Source Code

Condition.h

```
#ifndef Condition
#define ConditionH
#include <string>



namespace Optimizer
{


class Condition
/// This is a pure virtual class for conditions to be met under evaluation.
/// Subclasses can be added as needed to support different types of conditions
/// and different types of output files to read.
/// Two initial subclasses have been written for BTEC, one for a condition to
/// be maximized anywhere on the domain, and one for an condition not to be
/// exceeded.
/// Last revision: 1/26/09, A.W. Bailey.

{
protected:
        std::string mOutputFile; /// Filename of output file examined
        double mGoal;  /// Goal value

public:
        /// Constructor
        Condition() {}

        /// Destructor
        ~Condition() {}


        /// Initializer
        void init(std::string outputFile, double goal)
        {
                mOutputFile = outputFile;
                mGoal = goal;
        }

        /// Note if is a constraint
        virtual bool isConstraint() = 0;

        /// Look at the file and see how well the condition is met
        virtual double getBadness() = 0;

};

} // End namespace optimizer
#endif
```

HELCAT_Utility.h

```
#ifndef UtilityH
#define UtilityH
#include <cstring>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#include <algorithm>
#include <stdlib.h>
#include <string>
#include <sstream>
#include <ctime>


// The following #define _USE_MATH_DEFINES is for Visual C++ foible
#define _USE_MATH_DEFINES
//#include <math.h>
#include <cmath>

//#include "ConsoleLasem_stdafx.h"

//#include <osg/Vec3d>
//#include <osg/Matrix>

// Visual C++ precompiler "loses" the global math constants, so we put the
// ones we need here
#define M_PI       3.14159265358979323846
#define M_PI_2     1.57079632679489661923

/// \file
/// This file is used to hold a number of templates and function
/// that are of general use.  It also declares all the usual libraries.
/// By including it as a header file, most other files standardly needed
/// are included.
/// Math-Utility level

namespace HELCAT_tk
{
/*
HELCAT_TK_2_API double RadiansToDegrees( double a_angle_radians);
HELCAT_TK_2_API double DegreesToRadians( double a_angle_degrees);

class Vector;
HELCAT_TK_2_API osg::Vec3d ConvertToOSG( const Vector & a_vector_HELCAT);
class Position;
HELCAT_TK_2_API osg::Matrix ConvertToOSG( const Position & a_matrix_HELCAT);
*/

/// Transfer of sign
template <class T>

inline  T sign ( T a,  T b) {
    return (a < 0 ? -b : b);
};


#ifdef USEFASTMATH
#include <fastmath>

/// Get sin and cosine from fastmath routines
inline void sincos(double a, double &x, double &y)
{
  _fm_sincos(a, &x, &y);
}

/// Get sin and cosine from fastmath routines
inline void sincosl(long double a, long double &x, long double &y)
{
```

```
  _fm_sincosl(a, &x, &y);
}

#endif
#ifndef USEFASTMATH


/// Get sin and cosine from regular routines
inline void sincos(double a, double &x, double &y)
{
    x = sin(a);
    y = cos(a);
}

/// Get sin and cosine from regular routines
inline void sincosl(long double a, long double &x, long double &y)
{
    x = sinl(a);
    y = cosl(a);
}

#endif

/// Get the current date in iso format yyyy-mm-dd
/// This routine gives a warning in Visual C++
/// even though the code is ANSI C++ complient.
#pragma warning (disable : 4996)
inline std::string getDate()
{
        time_t raw_time;
        time(&raw_time);
        struct tm * timeinfo = localtime(&raw_time);
        char buffer[25];
        strftime(buffer,25, "%Y-%m-%d", timeinfo);
        return (std::string(buffer));
}


/// Search through an array for an interval in which a value resides
template <class T>
inline int hunt(const T value, T *array, int narray)
{
    // Hunt through to find the interval in which a value resides
    int nlow = 0;
    int nhigh = narray;
    int nmid;
    while ((nhigh - nlow) > 1)
    {
        nmid = (nhigh + nlow)/2;
        if (array[nmid] > value) nhigh = nmid;
        else nlow = nmid;
    }
    return nlow;
};


/// Set a object from a string
template <class T>
inline void fromstring(std::string s, T & x)
{
    std::istringstream iss(s);
    iss >> x;
       if (iss.fail())
               throw std::invalid_argument("Bad conversion from string");
};

// Specialization for objects that are already strings
template<>
inline void fromstring(std::string s, std::string& x)
{x = s;}
```

```
/// Convert an object of type T to a string
template <class T>
inline std::string tostring(const T& t, int precision)
{
    std::ostringstream os;
    os << std::showpoint << std::setprecision(precision) << t;
    return os.str();
}

template<class T>
inline std::string tostring(const T& t)
{
        std::ostringstream os;
        os << std::showpoint << std::setprecision(7) << t;
        return os.str();
}

/// Specializations for objects that are already strings
template<>
inline std::string tostring(const std::string& t)
{return(t);}

/// Compute the square of a value
template <class T>
inline T square(const T x)
{
    return x*x;
}

/// Compute the cube of a value
template <class T>
inline T cube(const T x)
{
    return (x*x*x);
}

/// Swap the values of two variables
template <class T>
inline void swap(T & x, T & y)
{
        const T z = x;
        x = y;
        y = z;
}


/// Delete any data associated with an array pointer
template <class T>
inline void clearPointer ( T*& p)
{
    if (p != 0)
    {
        delete[] p;
        p = 0;
    }
}

/// Return a new line character as a string
inline std::string newLine()
{
    char nl[2];
    nl[0] = '\n';
    nl[1] = 0;
    std::string nls = std::string(nl);
    return nls;
}

/// Insert a number between to strings with no additional blanks
inline std::string insertNumber(std::string beginning, int number, std::string ending)
{
        std::ostringstream oss;
```

```
        oss << beginning << number << ending;
        return oss.str();
}


}

#endif
```

Optimizer.h

```
#ifndef OptimizerH
#define OptimizerH

#include "mpi.h"
#include "stdafx.h"
#include "SpecificConditions.h"
#include "helcat_utility.h"
#include "mathexpr.h"
#include "pgapack.h"
#include <vector>
#include <iostream>
#include <fstream>



namespace Optimizer
{

        const static int MAXVARS = 10;
        const static int MAXFILES = 10;

        /// Variables for optimization
        double optVar[MAXVARS];

        /// Pointers to mappings between variable strings and values
        RVar* optVarMap[MAXVARS];

        double optVarMin[MAXVARS]; ///< Minimum values for optimizer variables
        double optVarMax[MAXVARS]; ///< Maximum values for optimizer variables

        double BADFAIL = 1.0E10; ///< Failure flag for out of range for variables

        /// File holding the variable files that are to be altered, for this process
        std::string varInProcFile[MAXFILES];

        /// Files holding the initial value of the input file strings
        std::string varInputValue[MAXFILES];

        /// Name of the executable string to be used: filename and arguments for this process
        std::string executeString;

        /// String used to create a directory for this process
        std::string mkdirString;

        /// String used to remove the directory for this process
        std::string rmdirString;

        /// Number of variables to be replaced (named %Var0%, %Var1%, etc.)
        int numOptVars;

        /// Number of alterable input files to use for the executed program,
        /// including the main configuration file
        int numVarInFiles;

        /// Output file for optimizer
        std::string optOutFile;

        /// Condition requirements.  Pointer to conditions.
        /// Boost library shared pointers would be better here,
        /// but sticking to standard library for now.
        /// Conditions must be set up by the calling routine and not deleted until
        /// they are no longer needed.
        /// This library only stores pointers to the conditions.
        std::vector<Condition*> optConditions;

        /// Evaluate the goodness of a function for a particular set of parameters
        /// The return value is zero when conditions are perfect
        double evaluateCase(PGA::PGAContext *ctx, int p, int pop);
```

```
        /// This routine takes a filename and mutates it based on the processor rank
        std::string makeProcName(std::string path, std::string filename, int rank);

        /// This routine replaces all occurrences of a substring in in a string
        /// with a replacement substring
        std::string findAndReplace(std::string oldstring, std::string oldsubstring,
                std::string newsubstring);

        /// This function reads in an input file into a string
        /// It strips off ending comments
        /// This eliminates any % signs in the comments that might cause problems
        std::string readInputFile(std::string inFile);

        /// This routine is used to flip slashes from forward to back for Windows
        std::string flipslash(std::string a_string);

        /// Routine to print the current state of the optimization
        void printCurrentState(PGA::PGAContext *ctx, int pop);

        /// Routine to print the final state of teh optimization
        void printFinalState(PGA::PGAContext *ctx, int pop);

        /// This routine constructs the execution string for the specific program
        std::string getExecutionString(std::string* input_files,
                std::string output_directory);

        /// Routine to clean input lines of comments
        std::string cleanString(std::string linestring);

        /// Routine to set the name of the output file read by a condition
        std::string getConditionOutputFile(std::string filename,
                std::string output_directory);

} // End namespace

namespace PGA
{

        /// Routine to replace PGARun
        void MyPGARun(PGA::PGAContext *ctx);

}

#endif
```

Optimizer.cpp

```
// Optimizer.cpp : Defines the entry point for the console application.
//

#include "Optimizer.h"
#include "KeyValueFileReader.h"
#include <iomanip>
#include <sstream>

using namespace std;
using namespace Optimizer;


/// This optimizer code uses the Argonne National Laboratories Parallel Genetic
/// Algorithm PGAPack library.  The code is designed to call BTEC (or any other
/// code, writing new input files and reading new output files.
/// Last revision: 1/26/09, A. W. Bailey.



        int main(int argc, char** argv)
{

        PGA::PGAContext *ctx;

        int myrank; /// Processor rank
        kvfr::KeyValueFileReader fileReader;  ///< Key-value file reader

        int maxIterations; ///< Maximum number of iterations to allow
        int generationSize; ///< Number of individuals in each generation
        int randomNumberSeed; ///< Random number seed (defaults to 1)
        int numReplace; ///< Number of individuals to replace each generation
                                    ///< (default is number of processors - 1)
        int numConditions; ///< Number of conditions to be satisfied
        string varInFile[MAXFILES]; ///< Input files with variable values
        string workPath; ///< Path for temporary work files


        // Initialize MPI
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);


        // Read in the input file
        string optInput = argv[1];
        ifstream optimizer_input(argv[1],ios::in);
        fileReader.init(optimizer_input);

        // Read in the maximum number of interations
        if(fileReader.setVar(maxIterations, "maxIterations"))
                maxIterations = 1000; // Default

        // Read in the generation size
        if (fileReader.setVar(generationSize,"generationSize"))
                generationSize = 100; // Default

        // Read in the random number seed
        if (fileReader.setVar(randomNumberSeed,"randomNumberSeed"))
                randomNumberSeed = 1; // Default

        if (fileReader.setVar(numReplace, "numReplace"))
        {
                // Base the number to replace each generation on the number
                // of processors being used
                int num_processes;
                MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
                if (num_processes <= 2)
                {
```

```
                        numReplace = num_processes;
                }
                else
                {
                        numReplace = num_processes - 1;
                }
        }

        // Read the name of the output file and open it
        fileReader.setVar(optOutFile, "optOutFile");

        // Open and close the file to get rid of any previous values
        ofstream optOutStream;
        optOutStream.open(optOutFile.c_str(), ios::trunc);
        optOutStream.close();

        // Read in the number of varying input files for the executable
        fileReader.setVar(numVarInFiles, "numVarInFiles");

        // Read in the names of the varying input files for the executable
        for (int n=0; n< numVarInFiles; n++)
        {
                fileReader.setVar(varInFile[n],
                        HELCAT_tk::insertNumber("varInFile[", n, "]"));
        }

        // Read in the work file path.  Remove any "/" at the end.
        fileReader.setVar(workPath, "workPath");
        int pathSize = int(workPath.size()) -1;
        if (workPath[pathSize - 1] == '/')
                workPath = workPath.substr(0, pathSize-1);

        // Read in the files with variables to be changed
        for (int n=0; n<numVarInFiles; n++)
        {
                varInputValue[n] = readInputFile(varInFile[n]);
        }

        // Create the strings for creating and removing work directories for
        // this process
        string dirString = workPath + string("/Process")
                + HELCAT_tk::tostring(myrank) + string("Temp");
#ifdef WIN32
                mkdirString = string("mkdir ")
                        + flipslash(dirString);
                rmdirString = string("rmdir ")
                        + flipslash(dirString) + string (" /s /q");
#else
                mkdirString = string("mkdir ") + dirString;
                rmdirString = string("rm -r ") + dirString;
#endif

        // Construct the variable input files needed based on which
        // processor is being used
        for (int n=0; n<numVarInFiles; n++)
        {
                varInProcFile[n] = dirString +
                        HELCAT_tk::insertNumber("/Varfile", n, ".txt");
        }

        // Replace the names of the variable input files
        for (int n=0; n<numVarInFiles; n++)
        {
                for (int i=0; i<numVarInFiles; i++)
                {
                        varInputValue[n] = findAndReplace(
                                varInputValue[n],
                                HELCAT_tk::insertNumber("%File", i, "%"),
                                varInProcFile[i]);
                }
                // Write out the new input file
```

```
                ofstream var_out(varInProcFile[n].c_str(),ios::trunc);
                var_out << varInputValue[n];
                var_out.close();
        }

        // Construct the executable file string
        executeString = getExecutionString(varInProcFile, dirString);

        // Get the number of variables to be optimized
        fileReader.setVar(numOptVars,"numOptVars");
        for (int i=0; i<numOptVars; i++)
        {
                // Get the max and min values for the variables
                fileReader.setVar(optVarMin[i],
                        HELCAT_tk::insertNumber("optVarMin[", i, "]"));
                fileReader.setVar(optVarMax[i],
                        HELCAT_tk::insertNumber("optVarMax[", i, "]"));

                // Set up mapping of the variable strings and value
                string varname = string("Var") + HELCAT_tk::tostring(i);
                optVarMap[i] = new RVar(varname.c_str(),
                        &optVar[i]);
        }

        // Get the optimization condition and constraints
        fileReader.setVar(numConditions,"numConditions");
        for (int i=0; i<numConditions; i++)
        {
                string condition_type;
                double condition_goal;
                string conditionOutputFile;
                Condition* new_condition;
                fileReader.setVar(condition_goal,
                        HELCAT_tk::insertNumber("condition[", i, "]_goal"));
                fileReader.setVar(conditionOutputFile,
                        HELCAT_tk::insertNumber("condition[", i, "]_outputFile"));

                // Pick the appropriate condition object to use

                // Don't need an ifdef here:  must be at least one condition
                if ( i == 0 ){new_condition = new CONDITION0 ;}

#ifdef CONDITION1
                else if (i == 1) {new_condition = new CONDITION1 ;}
#endif
#ifdef CONDITION2
                else if (i == 2) {new_condition = new CONDITION2 ;}
#endif
#ifdef CONDITION3
                else if (i == 3) {new_condition = new CONDITION3 ;}
#endif
#ifdef CONDITION4
                else if (i == 4) {new_condition = new CONDITION4 ;}
#endif
#ifdef CONDITION5
                else if (i == 5) {new_condition = new CONDITION5 ;}
#endif
#ifdef CONDITION6
                else if (i == 6) {new_condition = new CONDITION6 ;}
#endif
#ifdef CONDITION7
                else if (i == 7) {new_condition = new CONDITION7 ;}
#endif
#ifdef CONDITION8
                else if (i == 8) {new_condition = new CONDITION8 ;}
#endif
#ifdef CONDITION19
                else if (i == 9) {new_condition = new CONDITION9 ;}
#endif

                else
```

```
                {
                        // Unknown condition type
                        throw invalid_argument("Unknown condition type");
                }

                // Change condition output file, depending on processor
                conditionOutputFile= getConditionOutputFile(
                        conditionOutputFile, dirString);
                new_condition->init(conditionOutputFile, condition_goal);
                optConditions.push_back(new_condition);
        }

        // Set up PGA optimizer
        ctx = PGA::PGACreate(&argc, argv, PGA_DATATYPE_REAL,
                numOptVars, PGA_MINIMIZE);

        // Use random seed
        PGA::PGASetRandomSeed(ctx, randomNumberSeed);

        // Set maximum number of iterations and stopping criteria
        PGA::PGASetStoppingRuleType(ctx, PGA_STOP_MAXITER);
        PGA::PGASetStoppingRuleType(ctx, PGA_STOP_TOOSIMILAR);
        PGA::PGASetStoppingRuleType(ctx, PGA_STOP_NOCHANGE);
        PGA::PGASetMaxGAIterValue(ctx, maxIterations);



        // Set the population size and replacement criteria
        PGA::PGASetPopSize(ctx,generationSize);
        PGA::PGASetNumReplaceValue(ctx, numReplace);



        // Set to print each generation and to print the evaluation string
        PGA::PGASetPrintFrequencyValue(ctx, 1);
        PGA::PGASetPrintOptions(ctx, PGA_REPORT_STRING);


        // Set the maximum and minimum limits on variables
        PGA::PGASetRealInitRange(ctx, optVarMin, optVarMax);

        // Set the crossover type from the default with using only two variables
        if (numOptVars == 2)
                PGA::PGASetCrossoverType(ctx, PGA_CROSSOVER_ONEPT);

        // Run the optimizer
        PGA::PGASetUp(ctx);
        PGA::MyPGARun(ctx);


        // Finish up
        for (int n=0; n<numOptVars; n++)
        {
                // Delete mapping variables
                delete optVarMap[n];
        }
        for (unsigned int n=0; n<optConditions.size(); n++)
        {
                // Delete condition objects
                delete optConditions[n];
        }

        PGA::PGADestroy(ctx);
        MPI_Finalize();
        exit(0);
}


namespace Optimizer
{
```

```
/// Evaluate the goodness of a function for a particular set of parameters
/// The return value is zero when conditions are perfect
double evaluateCase(PGA::PGAContext *ctx, int p, int pop)
{
        double badness; // Measure of how good the solution is not

        // Construct a directory for files used in evaluating this case
        system(mkdirString.c_str());

        // Try to evaluate the case
        try
        {
                for (int i=0; i<numOptVars; i++)
                {
                        // Get the new value to be used
                        double newval = PGA::PGAGetRealAllele(ctx, p, pop, i);
                        optVar[i] = newval;

                        // Check if out of range
                        if ((optVar[i] < optVarMin[i]) || (optVar[i] > optVarMax[i]))
                                throw runtime_error("Variables out of range");
                }

                // Replace the varying values in the input files
                for (int i=0; i<numVarInFiles; i++)
                {
                        string input_now = varInputValue[i];
                        string::size_type startIndex = 0;
                        while (true)
                        {
                                // Find the next value to replace
                                startIndex = input_now.find("%",startIndex);
                                if (startIndex == string::npos)
                                        break; // Break if done

                                string::size_type endIndex =
                                        input_now.find("%",startIndex+1) + 1;

                                // Pull off the expression between the two %s
                                string expression = input_now.substr(startIndex+1,
                                        (endIndex - startIndex) -2);

                                // Set the proper values
                                // Evaluate the expression using the current values
                                char s[100];
                                strcpy(s,expression.c_str());
                                ROperation op(s, numOptVars, optVarMap);
                                double var_value = op.Val();

                                // Replace the placeholder with the appropriate value
                                input_now.replace(startIndex, endIndex - startIndex,
                                        HELCAT_tk::tostring(var_value));
                        }

                        // Write out the new file
                        ofstream exec_input(varInProcFile[i].c_str(),ios::trunc);
                        exec_input << input_now;
                        exec_input.close();
                }

                // Execute the worker program
                int flag = system(executeString.c_str());
                if (flag != 0)
                {
                        // Program did not execute properly.  The most likely reason is
that
                        // the optimizer-produced inputs are toxic.  Treat as a bad result.
                        throw runtime_error("Program did not execute properly");
                }
```

```
                        // Set the goodness value based on all of the conditions
                        badness = BADFAIL;
                        for (unsigned int n=0; n<optConditions.size(); n++)
                        {
                                Condition& cond = *(optConditions[n]);
                                if (cond.isConstraint())
                                {
                                        // Abort the the constraint is not satisfied
                                        double failflag = cond.getBadness();
                                        if (failflag > 0.0)
                                        {
                                                // Return sufficiently bad failure to exclude this
solution
                                                        badness = BADFAIL;
                                                        break;
                                        }
                                }
                                else
                                {
                                        // Set the badness of the solution
                                        badness = cond.getBadness();
                                }
                        }
                }
                catch(...)
                {
                        //Failure
                        badness = BADFAIL;
                }

                // Remove the directory
                system(rmdirString.c_str());

                // Return the evaluation
                return badness;
        }


        string makeProcName(string path, string filename, int rank)
                /// This routine takes a filename and mutates it based on the processor rank
        {
                // Find where the extension starts
                int dot_point = int(filename.rfind('.'));

                // Separate the file int a beginning and ending part
                string beginning = filename.substr(0, dot_point);
                int filesize = int(filename.size());
                string ending = filename.substr(dot_point, filesize - dot_point);

                // Construct a new filename with the path and with the rank value inserted
                string procFilename =  path + string("/")
                         + HELCAT_tk::insertNumber(beginning, rank, ending);
                return (procFilename);
        }

        /// This routine replaces all occurrences of a substring in in a string
        /// with a replacement substring
        string findAndReplace(string oldstring, string oldsubstring,
                string newsubstring)
        {
                string newstring = oldstring;
                string::size_type startpoint = 0;
                while(true)
                {
                        startpoint = newstring.find(oldsubstring, startpoint);
                        if (startpoint == string::npos)  break;
                        newstring.replace(startpoint, oldsubstring.size(),
                                newsubstring);
                        // Keep moving forward so that the routine does not break
                        // if the old and new substring values are identical
```

```
                        startpoint += newsubstring.size();
                }
                return newstring;
        }


/// This function reads in an input file into a string
/// It strips off ending comments
/// This eliminates any % signs in the comments that might cause problems
string readInputFile(string inFile)
{
        string outstring;
        ifstream varInput(inFile.c_str(),ios::in);
        // Read in the file, one line at a time
        while (varInput.good())
        {
                string linestring;
                getline(varInput, linestring);



                // Add line to new string, with end line
                outstring += cleanString(linestring) + HELCAT_tk::newLine();
        }
        return (outstring);
}


        /// This routine is used to flip slashes from forward to back for Windows
        string flipslash(string a_string)
        {
                // Go through character by character
                for (unsigned int n=0; n< a_string.size(); n++)
                {
                        if (a_string[n] == '/')
                                a_string[n] = '\\';
                }
                return a_string;
        }



/// Routine to print the current state of the optimization
void printCurrentState(PGA::PGAContext *ctx, int pop)
{
        static int generation_number = -1;

        // Set up string stream
        ostringstream oss;

        // Set output stream flags);
        oss << resetiosflags(ios::adjustfield);
        oss << setiosflags(ios::right);
        oss << setprecision(5);


        ofstream optOutStream;

        // New generation
        generation_number++;

        if (generation_number == 0)
        {
                // Open the output file, deleting any previous values
                optOutStream.open(optOutFile.c_str(), ios::trunc);

                // Print out column headings
                oss << setw(7) << "Gen" << setw(12) <<"Miss";
                for (int n=0; n< numOptVars; n++)
                {
```

```
                        string varOut = string("Var ") + HELCAT_tk::tostring(n);
                        oss << setw(12) << varOut;
                }
                oss << endl;
        }
        else
        {
                // Open the output file and append
                optOutStream.open(optOutFile.c_str(), ios::app);
        }


        // Print out the best answer so far
        int bestindex = PGA::PGAGetBestIndex(ctx, pop);
        double unfitness = PGA::PGAGetEvaluation(ctx, bestindex, pop);
        oss << setw(7) << generation_number << setw(12) << unfitness;
        for (int n=0; n<numOptVars; n++)
        {
                // Print out the current value of each varying input
                double varValue = PGA::PGAGetRealAllele(ctx, bestindex, pop, n);
                oss << setw(12) << varValue;
        }
        // End line and flush
        oss << endl;

        // Dump both to cout and optOutStream
        optOutStream << oss.str();
        cout << oss.str();
}

/// Routine to print the final state of the optimization
void printFinalState(PGA::PGAContext *ctx, int pop)
{
        // Open the output file and append this
        ofstream optOutStream;
        optOutStream.open(optOutFile.c_str(), ios::app);

        // Set output stream flags
        optOutStream << resetiosflags(ios::adjustfield);
        optOutStream << setiosflags(ios::right);
        optOutStream << setprecision(5);

        // Print out all of the final population
        optOutStream << "\n \n"
                << "Final population \n";
                // Print out column headings
                optOutStream << setw(7) << "Index" << setw(12) << "Miss";
        for (int n=0; n< numOptVars; n++)
        {
                string varOut = string("Var ") + HELCAT_tk::tostring(n);
                optOutStream << setw(12) << varOut;
        }
        optOutStream << endl;

        // Print out each individual in the population
        int population = PGA::PGAGetPopSize(ctx);
        for (int i=0; i<population; i++)
        {
                double unfitness = PGA::PGAGetEvaluation(ctx, i, pop);
                optOutStream << setw(7) << i << setw(12) << unfitness;
                for (int n=0; n<numOptVars; n++)
                {
                        // Print out the current value of each varying input
                        double varValue = PGA::PGAGetRealAllele(ctx, i, pop, n);
                        optOutStream << setw(12) << varValue;
                }
                optOutStream << "\n\0";
        }
        /// Flush
        optOutStream << endl;
}
```

```
}  // End namespace Optimizer


namespace PGA
{


        /// This a an version of PGARunGM, altered to provide for better I/0.
        void MyPGARun(PGAContext *ctx)
        {
                int       rank, Restarted;
                MPI_Comm comm;
                void    (*CreateNewGeneration)(PGAContext *, int, int);

                /*  Let this be warned:
                *  The communicator is NOT duplicated.  There might be problems with
                *  PGAPack and the user program using the same communicator.
                */
                //PGADebugEntered("MyPGARun");

                comm   = PGAGetCommunicator(ctx);
                rank = PGAGetRank(ctx, comm);

                PGAEvaluate(ctx, PGA_OLDPOP, Optimizer::evaluateCase, comm);
                if (rank == 0)
                        PGAFitness(ctx, PGA_OLDPOP);

                if (PGAGetMutationOrCrossoverFlag(ctx))
                        CreateNewGeneration = PGARunMutationOrCrossover;
                else
                        CreateNewGeneration = PGARunMutationAndCrossover;

                while (!PGADone(ctx, comm)) {
                        if (rank == 0) {
                                Restarted = PGA_FALSE;
                                if ((ctx->ga.restart == PGA_TRUE) &&
                                        (ctx->ga.ItersOfSame % ctx->ga.restartFreq == 0)) {
                                                ctx->ga.ItersOfSame++;
                                                Restarted = PGA_TRUE;
                                                PGARestart(ctx, PGA_OLDPOP, PGA_NEWPOP);
                                } else {
                                        PGASelect(ctx, PGA_OLDPOP);
                                        CreateNewGeneration(ctx, PGA_OLDPOP, PGA_NEWPOP);
                                }
                        }
                        MPI_Bcast(&Restarted, 1, MPI_INT, 0, comm);

                        PGAEvaluate(ctx, PGA_NEWPOP, Optimizer::evaluateCase, comm);
                        if (rank == 0)
                                PGAFitness(ctx, PGA_NEWPOP);

                        /*  If the GA wasn't restarted, update the generation and print
                        *  stuff.  We do this because a restart is NOT counted as a
                        *  complete generation.
                        */
                        if (!Restarted) {
                                PGAUpdateGeneration(ctx, comm);
                                if (rank == 0)
                                        Optimizer::printCurrentState(ctx, PGA_OLDPOP);
                                //PGAPrintReport(ctx, stdout, PGA_OLDPOP);
                        }
                }

                if (rank == 0) {
                        Optimizer::printFinalState(ctx, PGA_OLDPOP);
                        //best_p = PGAGetBestIndex(ctx, PGA_OLDPOP);
                        //printf("The Best Evaluation: %e.\n",
                        //      PGAGetEvaluation(ctx, best_p, PGA_OLDPOP));
                        //printf("The Best String:\n");
                        //PGAPrintString(ctx, stdout, best_p, PGA_OLDPOP);
```

```
                        //fflush(stdout);
                }
                //PGADebugExited("MyPGARun");
        }

}  // End namespace
```

ProgramSpecific.cpp

```
/// This file holds certain functions that must be written specifically
/// for the code

#include <string>
using namespace std;

namespace Optimizer{


/// This routine must be specifically written for each program to be optimized
/// The routine is passed the output directory to be used, in clase this is
/// one of the arguements to the program.  If it is not, then the output
/// directory will need to be included somewhere in the input file data.
/// Note that the output directory will be different for each process
/// when multiprocessing is used for optimization.
string getExecutionString(string* input_files, string output_directory)
{
        string prefixString = output_directory + string("/TEMP");
        string executionString = string("BTECThermal ") +
                input_files[0] + string(" ") + prefixString;
        return (executionString);
}

/// Construct the name of the output file for a condition, given the desired
/// file name and the output file directory.  This routine needs to be
/// rewritten for each program to be optimized.
string getConditionOutputFile(string filename, string output_directory)
{
        string prefixString = output_directory + string("/TEMP");
        string conditionOutputFilename = prefixString + string(".") + filename;
        return (conditionOutputFilename);
}


/// In order to perform substitutions properly, the code must not include
/// and percent signs that are not used for substitutions.  The BTEC code
/// frequently has these in comments, and this routine removes all comments
/// from a BTEC input file
string cleanString(string linestring)
{
        // Strip off any ending comments
        string::size_type commentIndex = linestring.find("#");
        if (commentIndex != string::npos)
        {
                // Strip off comment
                linestring = linestring.substr(0,commentIndex);
        }
        return (linestring);
}

}  // End namespace
```

SpecificConditions.h

```
#ifndef SpecificConditionsH
#define SpecificConditionsH
#include "Condition.h"

/// This header is used to define classes that are specific to the
/// program being optimized.  A new version must be constructed for
/// each program to be optimized.

#define CONDITION0 MaxAllowedDamage
#define CONDITION1 MaximizeTempRise

namespace Optimizer {

class MaxAllowedDamage : public Condition
        /// This represents the Condition class which sets a maximum allowed value
        /// for some variable.   The implimentation is specific to the probe files
        /// of the BTEC code.
{
public:
        // Constructor
        MaxAllowedDamage() {}


        bool isConstraint()
        {
                return (true);
        }

        double getBadness();
};

class MaximizeTempRise : public Condition
        /// This represents the Condition class which is optimal with the maximum
        /// peak value.   The implimentation is specific to the probe files of the
        /// BTEC code.
{
public:
        // Constructor
        MaximizeTempRise() {}

        bool isConstraint()
        {
                return (false);
        }

        double getBadness();
};

}  // End namespace

#endif
```

SpecificCondition.cpp

```
/// This file holds the routines for the specific conditions that are being
/// optimized for a program.
/// This file must be rewritten for the individual program to be optimized.

#include "SpecificConditions.h"
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>

using namespace std;

namespace Optimizer
{


/// Set whether a situation is within allowed boundaries
double MaxAllowedDamage::getBadness()
{
	// Open the output file
	ifstream outputStream(mOutputFile.c_str(), ios::in);

	// Read in each line and check its value
	while (outputStream.good())
	{
		string line;
		getline(outputStream,line);

		// Ignore comment lines
		if (line[0] == '#')
			continue;

		/// Get the value on the line
		double time, val;
		istringstream iss(line);
		iss >> time >> val;

		// Return if exceeds bounds
		if (val > mGoal)
			return (1.0);
	}

	// Reaching this point implies the condition was satisfied
	return (0.0);

}

/// Set whether a situation is within allowed boundaries
double MaximizeTempRise::getBadness()
{
	// Open the output file
	ifstream outputStream(mOutputFile.c_str(), ios::in);
	if (!outputStream.good())
		throw runtime_error("Bad read-in of output file");

	// Read in each line and check its value
	double minval = 1.0e10;
	double maxval = -1.0e10;
	while (outputStream.good())
	{
		string line;
		getline(outputStream,line);

		// Ignore blank and comment lines
		if (line.size() == 0)
			continue;
		if (line[0] == ' ')
```

```
                        continue;
                if (line[0] == '#')
                        continue;

                /// Get the value on the line
                double time, val;
                istringstream iss(line);
                iss >> time >> val;

                /// Update max an min values, if needed
            if (val > maxval)
            {
                    maxval = val;
            }
            if (val < minval)
            {
                    minval = val;
            }
        }

        // Return the badness as inverse of the difference between maximum and
        // minimum values
        return (1.0/(maxval - minval));

}

} // End namespace optimizer
```