JEGA, A Package for Single and Multiple Objective Genetic Optimization

Version 2.7 User's Manual

John P. Eddy, Ph.D.
System Readiness and Sustainment Technologies Department

Sandia National Laboratories
P.O. Box 5800 MS 1188
Albuquerque, New Mexico 87185

Abstract

JEGA is a software package that implements a multi-objective genetic algorithm (MOGA) for solution to multi-objective optimization problems (MOPs) and a single objective genetic algorithm (SOGA) for solution to single objective optimization problems (SOPs). JEGA is highly configurable and provides a rich set of algorithmic components for tailored optimization. It has been used successfully on many problems by users both internal and external to Sandia. By employing object-oriented design to implement abstractions of the key components making up the algorithms, JEGA provides a flexible and extensible problem-solving environment for design and performance analysis of computational models.

This report serves as a user's manual for the JEGA software and provides capability overviews and procedures for software execution, as well as a variety of example studies.

Acknowledgements

A number of people have contributed to the JEGA project in various ways. Special thanks go out to Dr. Laura Swiler at Sandia National Laboratories. She is ultimately responsible for the existence of JEGA and has provided and continues to provide valuable technical guidance.

Thanks go out to the rest of the DAKOTA development team who have all provided technical guidance during the development of JEGA.

Contents

Chapter 1

Introduction

The JEGA library contains two global optimization methods. The first is a Multi-objective Genetic Algorithm (MOGA) which performs Pareto optimization. This is the primary deliverable of the JEGA software. The second is a Single-objective Genetic Algorithm (SOGA) which performs optimization on a single objective function or a weighted sum of multiple objectives. Both methods support general constraints, a mixture of real and discrete variables, and a variety of objective types. The JEGA library was written by John Eddy, currently a principal member of the technical staff in the System Readiness and Sustainment Technologies department at Sandia National Laboratories, Albuquerque, New Mexico.

1.1 Motivation for JEGA Development

The following quote from the DAKOTA 5.3 User's Manual [**?**] briefly describes the motivation for optimization in general.

"Computational models are commonly used in engineering design and scientific discovery activities for simulating complex physical systems in disciplines such as fluid mechanics, structural dynamics, heat transfer, nonlinear structural mechanics, shock physics, and many others. These simulators can be an enormous aid to engineers who want to develop an understanding and/or predictive capability for complex behaviors typically observed in the corresponding physical systems. Simulators often serve as virtual prototypes, where a set of predefined system parameters, such as size or location dimensions and material properties, are adjusted to improve the performance of a system, as defined by one or more system performance objectives. Such optimization or tuning of the virtual prototype requires executing the simulator, evaluating performance objective(s), and adjusting the system parameters in an iterative, automated, and directed way. System performance objectives can be formulated, for example, to minimize weight, cost, or defects; to limit a critical temperature, stress, or vibration response; or to maximize performance, reliability, throughput, agility, or design robustness."

Of particular interest to the JEGA project are those problems that have multiple objectives. See Section 1.2 for a detailed discussion of multi-objective optimization problems (MOPs). Such problems are very commonly encountered in optimization. Quoting from [**?**]:

"Problems with multiple objectives arise in a natural fashion in most disciplines and their solution has been a challenge to researchers for a long time."

The solutions to multi-objective problems can provide a decision maker with a great deal of information about how the various objectives relate to one another including the trade-offs that must occur when choosing a final solution. This will be discussed further in Section 1.2 when the form of the solutions to these problems is discussed.

One of the primary motivations for the development of JEGA has been to provide engineers with a systematic means of obtaining improved or optimal designs using their simulator-based models. Making this capability available to engineers generally leads to better designs and improved system performance at earlier stages of the design phase, and eliminates some of the dependence on real prototypes and testing, thereby shortening the design cycle and reducing overall product development costs.

The next section discusses both single and multiple objective optimization problems in mathematical detail.

1.2 Optimization Problems

The standard form single objective optimization problem (SOP) is shown in Equation 1.1.

$$
\begin{aligned}
&\textit{minimize}: \\
&\quad F(\bar{x}) \\
&\\
&\textit{subject to}: \\
&\quad g_i(\bar{x}) \leq 0 \quad i = 1, 2, ..., m \\
&\quad h_i(\bar{x}) = 0 \quad i = 1, 2, ..., p \\
&\quad x_{li} \leq x_i \leq x_{ui}
\end{aligned}
\tag{1.1}
$$

Where $\bar{x}$ is a vector of design variables, F is a scalar function or composition of functions resulting in a scalar value, g are inequality constraints, h are equality constraints, and the variables may be bounded. The fact that F is scalar indicates that this is a single objective problem.

As stated, the primary deliverable of JEGA is a multi-objective genetic optimizer which operates on multi-objective optimization problems. The general form of an MOP is shown in Equation 1.2 below.

$$
\begin{aligned}
&\textit{minimize}: \\
&\quad \bar{F}(\bar{x}) = [f_1(\bar{x}), f_2(\bar{x}), ..., f_k(\bar{x})]^T \\
&\\
&\textit{subject to}: \\
&\quad g_i(\bar{x}) \leq 0 \quad i = 1, 2, ..., m \\
&\quad h_i(\bar{x}) = 0 \quad i = 1, 2, ..., p \\
&\quad x_{li} \leq x_i \leq x_{ui}
\end{aligned}
\tag{1.2}
$$

The difference between Equation 1.1 and Equation 1.2 is that in Equation 1.2, the objective function is vector valued.

When the objective function in a problem is vector valued, there is the potential for a vector valued solution. The existence of such a solution depends on the relationships that exists between the objectives. The three possible relationships between any two objectives are as follows:

1. Cooperative - This is the relationship that exists if the two objectives share design variables and are not in competition with one another. This means that the two objectives desire the same trends in the shared design variables and that improvement of one objective typically accompanies improvement of the other.
2. Competitive - This is the relationship that exists if the two objectives share design variables and are not cooperative with one another. This

means that the two objectives desire a different trend for at least one of the shared variables and that improvement of one typically accompanies worsening of the other.

3. Indifferent - This is the relationship that exists if the two objectives have no design variables in common. In this case, the two objectives move independently of one another. The problem may be effectively solved by carrying out two separate single objective optimizations depending on the form of any shared constraints or may be solved using an appropriate weighted-sum-of-objectives scheme.

The case of indifferent objectives is quite un-interesting from a multi-objective optimization viewpoint because it does not require true multi-objective optimization and it can generally be detected prior to any analysis or optimization. This case can be expressed using set notation as follows:

$$f_1(\bar{x}_1 \subseteq \bar{x}), f_2(\bar{x}_2 \subseteq \bar{x}) : \bar{x}_1 \cap \bar{x}_2 = \varnothing \tag{1.3}$$

where $\bar{x}$ is the set of all design variables used throughout the entire problem, $\bar{x}_1$ is the subset of $\bar{x}$ used by objective 1, and $\bar{x}_2$ is the subset of $\bar{x}$ used by objective 2. In this equation, $\bar{x}$ has components $\bar{x} = \{x_1, x_2, x_3 ... x_n\}$ where n is the total number of design variables.

The case in which all objectives are cooperative is more interesting than that of indifference because it is often not possible to detect this situation prior to the optimization. However, the result of solving a problem such as this will generally be a single superior solution or a set of solutions with the exact same superior performance characteristics.

The case in which two or more of the objectives in a multi-objective problem are in competition with one another is by far the most interesting from a multi-objective optimization viewpoint. It is in this case that the solution will no longer be a single superior design point but instead will be a set, possibly infinite in size, of efficient solutions called the *Pareto optimal set* [**?**].

The Pareto optimal set is the collection of all Pareto optimal solutions. A Pareto optimal solution is one that is efficient or non-dominated in the set of all possible solutions where the criteria for dominance is given in Equation 1.4 below.

A feasible performance vector $\bar{u} = \{u_1, u_2, u_3 ... u_k\}$ dominates a feasible performance vector $\bar{v} = \{v_1, v_2, v_3 ... v_k\}$ (denoted $\bar{u} \preceq \bar{v}$) iff $\bar{u}$ is partially less than $\bar{v}$.

$$\forall i \in \{1, ..., k\}, u_i \leq v_i \wedge \exists i \in \{1, ..., k\}, u_i < v_i \tag{1.4}$$

where k is the total number of objectives each of which is to be minimized.

Equation 1.4 states that in order for one design to dominate another it must be better with respect to at least one objective and no worse with respect to all others.

The Pareto optimal set is then given by Equation 1.5 below.

$$P^* := \{\bar{x} \in \Omega \mid \neg\exists \bar{x}' \in \Omega \quad \bar{F}(\bar{x}') \preceq \bar{F}(\bar{x})\} \tag{1.5}$$

where Ω is the set of all feasible solutions to the problem [**?**].

Equation 1.5 states that the Pareto optimal set is defined as the set of all feasible solutions for which there are no other feasible solutions that dominate them, or more simply, every solution that is non-dominated in the entire feasible space.

Plotting the Pareto optimal set in the performance space displays the Pareto frontier which is a portion of the boundary of the feasible space such that all points on the region are Pareto optimal as shown in Figure 1.1 below. The blue line in the figure is the Pareto frontier.

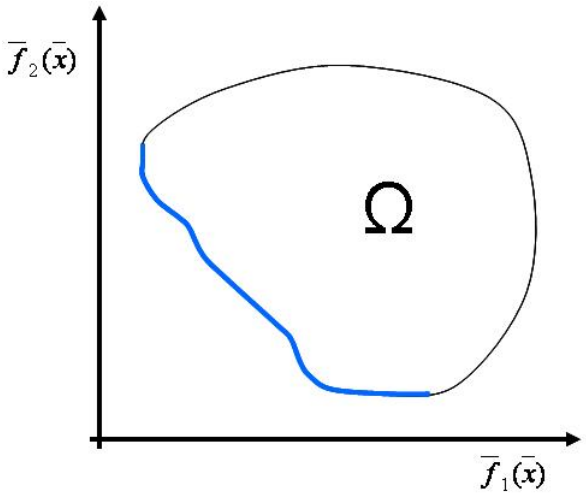

Figure 1.1: The Typical Looking Pareto Frontier.

As can be seen from the figure, having a representation of the Pareto frontier can not only provide a decision maker with a variety of possible efficient solutions, but it can also provide a designer with information about the trade-offs that exist between objectives. For example, with such a curve, a designer can answer questions like:

"From my current candidate solution, how much of objective 1 would I have to sacrifice to achieve a corresponding improvement of X% in objective 2?"

There is a vast body of research involving techniques for solving single objective optimization problems. The options for solving multiple objective problems are

considerably more limited. Problems such as these can be difficult to solve. Often, multi-objective optimization problems are converted into single objective problems for the purpose of solution. There are a number of ways to do this including various weighted sum schemes, the normal boundary intersection (NBI) method [**?**], goal programming, utility theoretical methods, etc. Generally, such techniques require the repeated solution of the resulting single objective problem using different problem parameters in order to generate a sampling of the Pareto optimal set. Each of these techniques suffers from certain drawbacks generally caused by a sensitivity to the shape of the Pareto frontier [**?**].

JEGA employs a genetic algorithm to solve these sorts of problems. the advantages of using evolutionary algorithms for solving such problems are many. They include:

- insensitivity to the shape of the Pareto frontier;
- intrinsic maintenance of a set of solutions;
- solution to the problem in a single optimization;
- potential for global solutions; and
- zeroth order operations.

There are also disadvantages to using evolutionary algorithms for solving optimization problems such as:

- a need for a large number of objective function and constraint function evaluations;
- no guarantee of optimality nor any indication of degree of optimality of a solution;
- no guarantee of achieving the same solution on subsequent runs of the algorithm.

1.3 Capabilities of JEGA

JEGA is highly configurable and is therefore very flexible both at run time and at compile time. In addition, it is easily extensible such that new algorithmic components can be inserted with few or no changes to the core code.

1.4 How Does JEGA Work

JEGA can easily be used as a library by other programs or as a stand alone application. The advantage to the stand alone approach is that only your evaluation code would have to be written and compiled instead of re-compiling and/or relinking JEGA. The disadvantage is that communication between JEGA and the evaluation code takes place through the file system which is many orders of magnitude slower than the direct interface possible when using JEGA as a library.

1.5 Using This Manual

Chapter 2

Background

JEGA is the evolution of work began in 1999 at the University at Buffalo, SUNY in Buffalo, NY while persuing a masters degree. At the time the package had no official name and contained only a multi-objective genetic algorithm. The operations are detailed in [**?**].

Over the next two years after completion of the masters thesis, the algorithm was refined and re-factored and built into a larger package as an optimization component. This work was also completed at SUNY Buffalo by John Eddy. This package was never made publicly available. It included a handful of optimization methods and a custom visualization technique called Cloud Visualization [**?**].

In the summer of 2003 while still working on a Ph.D. John was hired on at Sandia labs as a technical intern working in the Optimization and Uncertainty Estimation department. During the three month internship, the code was again re-factored and made a sub-package of The DAKOTA Project and was given the name JEGA. It was also at this time that a separate single objective GA was added to the package.

With the exception of minor bug fixes, the code available through DAKOTA remained the same until December 2006. During the interim time, John continued to modify, extend, and improve JEGA in support of his doctoral work. Once finished with his doctorate, John was hired on as a permanent member of the technical staff at Sandia in the System Readiness and Sustainment department. In the time since, JEGA has undergone a great deal of development but the software design has remained conceptually in tact. Many additional capabilities, bug fixes, new operator types, new operator specializations, etc. have been added.

In September 2006, JEGA was officially migrated out of the DAKOTA project and became an independent Sandia software development project. That is the current status of JEGA.

Chapter 3

Getting Started

I have a multi-objective optimization problem to solve, how do I use JEGA?

JEGA can be used in a number of ways. Regardless of how you wish to use it, there are some common things that must be done.

3.1 Obtaining JEGA

Currently, JEGA can only be publicly obtained through The DAKOTA Project. Downloading DAKOTA will result in a download of the full JEGA package as well.

3.2 Building JEGA

JEGA is primarily meant to be compiled into a static library. It is intended to be used from within an existing or newly created program. There is a way to use JEGA as a stand alone application and provide input via an input file. See ?? for more details. Regardless of how you are using JEGA, the first step is to compile it. This of course only need be done once and then the JEGA library can be used in multiple projects. JEGA uses an autoconf build harness for Unix style systems and Visual Studio .NET 2003, 2005, and 2008 solution and project files for Windows. Build JEGA with your desired configuration options and make note of the name and location of the resulting library file. See Chapter 5 for information on the configuration options.

3.2.1 Building on Unix and Unix-like Systems

The JEGA distribution includes the files produced by autoconf. The end user need only issue two commands to compile JEGA. The first is a call to the configure script. You will issue this call with any desired arguments as described in Section 5.1. It is strongly recommended that you maintain separate build and source trees. In order to accomplish this, you should issue the configure call from a directory other than `$(JEGA_ROOT)`. Recommended practice is to create subfolders within the `$(JEGA_ROOT)/build` directory. This way, you can maintain multiple configurations on multiple different platforms/machines, etc. simultaneously.

Once configuration is complete, JEGA can be built with a call to make. The commands that must be issued are shown below.

```
>> $(JEGA_ROOT)/configure ARGS
>> make
```

As an example, suppose you have multiple machines on which you may want to use JEGA. Suppose that the JEGA source distribution is on a shared drive and you wish to share a source tree but not a build tree. Also suppose that you wish to maintain both a debug and a release build. Here is the recommended procedure. Lets say the two machines are named yin and yang.

- Browse into `$(JEGA_ROOT)/build`:
```
>> cd $(JEGA_ROOT)/build
```

- Make a directory for each of yin and yang:
```
>> mkdir yin
>> mkdir yang
```

- Make subdirectories in each for debug and release:
```
>> cd yin
>> mkdir debug
>> mkdir release
>> cd ../yang
>> mkdir debug
>> mkdir release
```

- Enter the yin debug directory while on yin, configure, and make:
```
>> <LOGON> yin
>> cd $(JEGA_ROOT)/build/yin/debug
>> $(JEGA_ROOT)/configure CXXFLAGS=-g --enable-debugging
>> make
```

- Enter the yin release directory while on yin, configure, and make:

```
>> cd ../release
>> $(JEGA_ROOT)/configure CXXFLAGS=-O2
>> make
```

- Do the same in the yang directories while on yang.

Once you have completed these steps, you will have static libraries in each of the 4 directories built from the same shared source tree.

3.2.2 Building in Microsoft Visual Studio

JEGA is distributed with solution and project files for both Visual Studio .NET 2003 (7.1), 2005 (8.0), and 2008 (9.0). The solution files can be found in `$(JEGA_ROOT)/build/vc71`, `$(JEGA_ROOT)/build/vc80`, `$(JEGA_ROOT)/build/vc90` respectively. Open the desired ".sln" file. There are a number of build configurations available. Each is set up to build JEGA with different options. See Table **??** below for a listing of those options. Choose your desired build configuration and then build the project. The result will be a library against which you can link and an executable if building the configuration file front end.

3.2.3 Building in Eclipse Using CDT

3.3 Using JEGA Once Built

In order to get started using the created library, you will have to create a project. Exactly what this requires depends on what system you are developing on. If developing on Window for example, you will want to create a new Visual Studio project or a new nmake project. If on a unix type system, you will create a new makefile project, etc. If you already have a project that you want to start using JEGA in, you need only modify it slightly to start using JEGA.

Regardless of what system you are using, there are some common requirements for using JEGA.

Required Include Paths

You will have to add some paths to your list of include directories. This is typically done with a command line argument to your compiler such as -I or /I.

- `$(JEGA_ROOT)/include`
- `$(JEGA_ROOT)/eddy`

There may be additional include path requirements depending on what system you are using. See the sections below for specific requirements.

Windows does not come with an implementation of pthreads which JEGA uses if compiled with `JEGA_THREADSAFE` defined. To deal with this case, JEGA is distributed with a snapshot of the PThreads Win32 headers and libraries (http://sourceware.org/pthreads-win32/). If you need to use them, then you must also include the path:

- $(JEGA_ROOT)/eddy/threads/pthreads/include

Required Libraries

You will also have to link against the JEGA library created during the build. This is typically specified with two inputs to the compiler. One to specify a path to look in and one to specify the name of the library. The library created when building JEGA will be called libjega.$(LIB_EXT) where $(LIB_EXT) is the file extension for static libraries on your system; typically .a or .lib.

The easiest thing to do is to follow the examples in Chapter 8.

3.3.1 Using JEGA on Unix and Unix-like Systems

If on a unix like system, you will want to add the paths to the configuration headers that were created by autoconf. There are two paths needed for two headers. They are in the build tree, not the source tree. Consider `$(BUILD_DIR)` to be the path to the trunk of your build directory in the discussion below. In the example of Section 3.2.1, `$(BUILD_DIR)` would be one of `$(JEGA_ROOT)/build/yin/debug`, `$(JEGA_ROOT)/build/yang/release`, etc. The directories you will need to add are:

- `$(BUILD_DIR)`
- `$(BUILD_DIR)/eddy`

3.3.2 Using JEGA in Microsoft Visual Studio

Chapter 4

Features

See Chapter 5 for information on how to enable/disable and use the features described in this chapter.

4.1 Operator Abstraction

The operations performed by JEGA are broken up both conceptually and in actual software design such that each is performed by a specific class of operator. The operators are invoked in some order until some stopping and/or convergence criteria has been met. Each of the operations can be performed by any number of available operator specializations. For example, the initialization of a population can be performed by any of the available "Initializers". Chapter 5 details the different specializations available for each operator type. The current operator categories are shown in the list below in no particular order. The description accompanying each is not intended to imply in any way exactly how these operations are carried out, only their desired result or intent.

- **Main Loops** - Define the progression or order of operations of the algorithm as well as perform any relevant intermediate actions.
- **Initializers** - Create the initial population of candidate solutions.
- **Mutators** - Add random variation to a group of candidate solutions for the purposes of design space exploration.
- **Crossers** - Mate existing candidate solutions together to create new candidate solutions for the purposes of exploitation of observed good solutions.
- **Convergers** - Determine when the algorithm should stop and return its current result.

- **Evaluators** - Perform the evaluation of response functions including objectives and constraints with respect to a supplied set of design variable values.
- **Fitness Assessors** - Assess the fitness of evaluated candidate solutions with respect to one another.
- **Selectors** - Choose a subset of all current candidate solutions to become the next population.
- **Niche Pressure Applicators** - Encourage differentiation amongst the candidate solutions for the purposes of exploration. Also referred to simply as "Nichers".
- **Post Processors** - Perform whatever operations are desired on the final set of reported solutions prior to their return from the algorithm.

This design allows a great deal of flexibility in the way JEGA behaves. It also allows for easy incorporation of new techniques for performing the various operations.

4.2 Running Multiple Algorithms

JEGA is the project that houses the various genetic algorithms available. It is not entirely abstract in that there is some code that is external to the algorithms themselves. This code is primarily embodied by the front end sub-project. Using this front end code, multiple GA's can be created and used to solve multiple problems within a single application.

4.3 Thread Safety

JEGA can be configured at compile time to be thread aware. In this way, multiple instances of the JEGA algorithms can be safely run simultaneously from different threads within a program. See Chapter 5 for information on how to configure for this feature at compile time.

4.4 Logging

JEGA reports information to users via a logging capability. It can be configured to log to the console, text files, or both (see Section 5.1.2 for more information).

No matter what else is happening, if file logging is enabled, JEGA will create and write some information to a "global" log file. This is where messages generated by non-algorithmic components are written.

JEGA can be used in a number of ways and multiple algorithms may be run in a given program. If multiple algorithms are to be run, each algorithm can be configured to log to a separate file or each can log to the "global" log file or any combination.

Every message has a similar format as shown below.

$$< day >< time >< year >:< level > - < issuer >:< message >$$

An example would then be:

Fri Sep 01 08:59:29 2006: quiet- JEGA Front End: Random seed = 12345

JEGA will not overwrite previous logs. If a file exists with the name supplied to JEGA, JEGA will open the existing file and begin appending log entries into it. For this reason, it is important to archive or discard old log files when appropriate.

4.5 Portability and Interoperability

The core of JEGA is written in ANSI C++ without the use of any vendor specific or third party libraries. All source code required to make and use JEGA as a library is included with the distribution. It is regularly tested on a number of platforms including multiple Unix and Linux flavors as well as on Windows (both natively and using cygwin). For building JEGA on the *nix platforms, an autoconf harness is included in the distribution. For building natively on windows, Visual Studio projects for each of VS .NET 2003, 2005, and 2008 are provided.

To create a JEGA executable from code requires use of boost libraries.

For Windows users, JEGA is also distributed with a Managed front end. This is a separate (small) body of code that will operate JEGA from a project that uses the Microsoft managed extensions for C++. Using this body of code, it is possible to use JEGA directly from within any of the .NET languages using common language runtime support. JEGA is also distributed with a Visual Basic front end that takes advantage of this capability. Other front ends are planned for the future including those for the other .NET languages as well as one for JAVA (non-Microsoft specific).

Chapter 5

Configuring JEGA

There are two conceptually different types of configuration to be discussed in this chapter. The first involves how to configure the source code when compiling JEGA. The second is how to configure JEGA to perform operations the way you want at run time.

5.1 Compile Time Configuration Options

JEGA is distributed as source code and thus, it must be compiled prior to being used. The following sections describe the various compile time options that are available with JEGA and how to use them.

5.1.1 Compiler Specification

You can specify to the configure utility which C++ compiler to use by providing a value for the CXX variable on the command line. You can specify flags to be passed to the compiler by providing a value for the CXXFLAGS variable on the command line. For example, suppose your default compiler is g++ 3.4 and you wish to use 4.0 and build with debugging information. The configuration command line would look like:

```
>> $(JEGA_ROOT)/configure CXX=/usr/local/bin/g++4 CXXFLAGS=-g
```

5.1.2 Logging

As discussed in Section 4.4, JEGA is capable of logging status messages describing the workings of the various algorithmic components, the progression of the algorithm, etc. This behavior can be controlled in part during compilation

of JEGA. JEGA supports logging to the console window using the standard output stream (cout for C++ programmers) and to a file with a name that can be specified programmatically at configuration time.

By default, JEGA will do no logging whatsoever. To enable logging, the preprocessor constant `JEGA_LOGGING_ON` must be defined for your project. This is typically accomplished by modifying make files and/or project files. If you are using the autoconf harness distributed with JEGA then logging is on by default and can be disabled by passing the optional flag `--disable-logging` at configuration time.

If you wish to log to only one or the other of the console and file you can optionally define constants to disable the other. They are `JEGA_LOGGING_NO_CONSOLE` and `JEGA_LOGGING_NO_FILE` respectively. These constants must be defined along with `JEGA_LOGGING_ON`. Defining either of these without defining `JEGA_LOGGING_ON` will have no effect and no logging will occur. Defining both of these will result in no logging regardless of whether or not you've defined `JEGA_LOGGING_ON`. If using the autoconf harness, then the options `--enable-logging-no-console` and `--enable-logging-no-file` can be used to achieve the desired effect.

To summarize, if you wish to do no logging, do not define `JEGA_LOGGING_ON`. If you wish to log to both the console window and to a file, define `JEGA_LOGGING_ON`. If you wish to log to only the console window, define `JEGA_LOGGING_ON` and `JEGA_LOGGING_NO_FILE`. Finally, if you wish to log only to a file and not to the console window, define `JEGA_LOGGING_ON` and `JEGA_LOGGING_NO_CONSOLE`.

5.1.3 Debugging

If distributed under the LGPL, GPL, or certain other open source licenses, JEGA comes equipped with a very simple but powerful debugging capability. A great deal of source code exists in the JEGA project to support the use of this facility. This code checks for exceptional, unusual, or erroneous conditions. If such a condition is found, the debugging code causes an assertion failure along with a rudimentary scope trace. The same scope trace will appear if a signal is caught with a description of the signal if it is known. Figure 5.1 below shows what one might see if a segmentation fault occurs during JEGA execution.

This capability is disabled by default. To enable it, you must define the `JEGA_OPTION_DEBUG` preprocessor constant on your compiler command line or in your project files. If using the autoconf harness that is distributed with JEGA, then you may supply the optional `--enable-debugging` flag during the configuration step to enable debugging.

Compiling with this option will add a fair amount of additional code to your assembly and slow down your execution. It is therefore only recommended as a debugging tool if you are having problems.

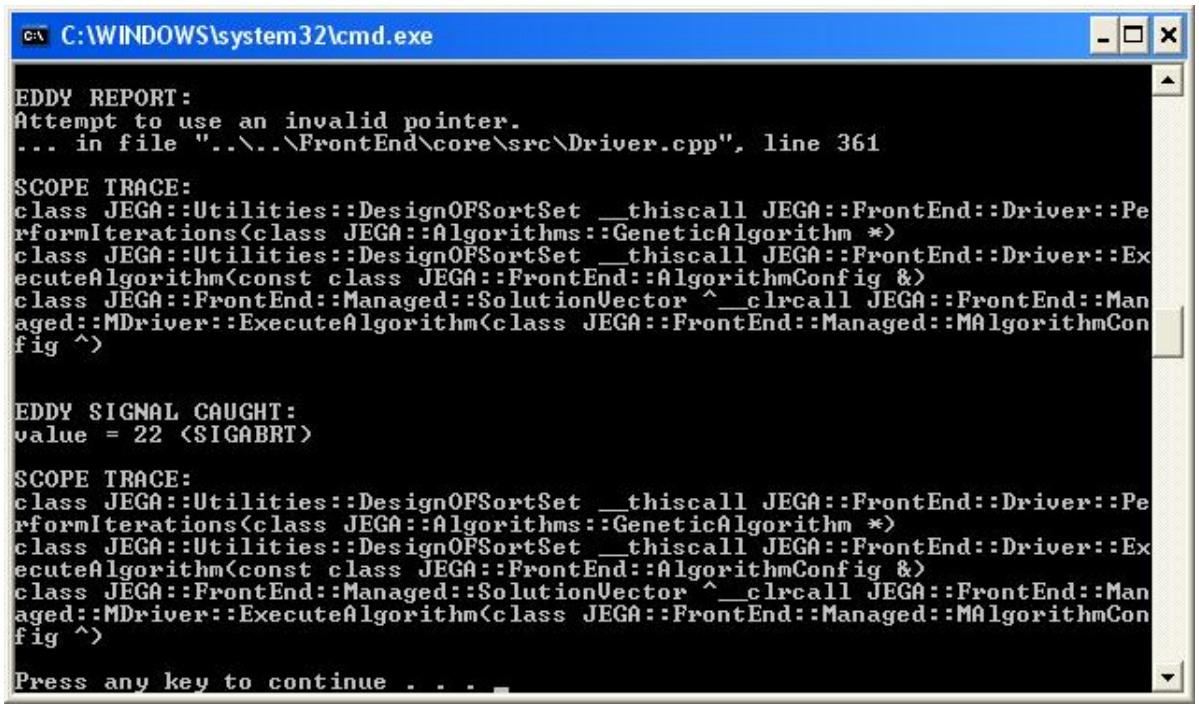

Figure 5.1: Example JEGA Debug Scope Trace

5.1.4 Thread Safety

As discussed in Section 4.3, JEGA can be configured to run in a thread safe manner. This allows multiple JEGA algorithms to be run in order to solve the same problem simultaneously from separate threads. This behavior can be controlled only during compilation of JEGA.

By default, JEGA will will not compile with thread safe behavior. To enable thread safety, the preprocessor constant `JEGA_THREADSAFE` must be defined for your project. This is typically accomplished by modifying make files and/or project files. If you are using the autoconf harness distributed with JEGA then thread safety can be enabled by passing the optional flag `--enable-threadsafe` at configuration time.

Note that this feature is not only relevant to the running of multiple concurrent algorithms, but also to the ability for JEGA evaluators to perform concurrent evaluations using multiple threads. If `JEGA_THREADSAFE` is not defined, concurrent evaluations cannot be performed.

5.1.5 Dynamically Linked Libraries

This discussion is meant specifically for Windows users since building shared objects in Unix and Unix-like environments is trivial in terms of source code modification. Windows requires modifications of the actual source code when building shared or dynamically linked libraries (dlls).

If you wish to compile JEGA into a dll, you must define certain constants to inform JEGA that it must insert the necessary code. Regardless of whether you are building an import or an export library, you must define `JEGA_SL` (for JEGA shared library). Then, you must define `JEGA_IMPORTING` or `JEGA_EXPORTING` depending on whether you are building an import or export library respectively.

There are no flags for this provided with the autoconf harness but there are project configurations available in the Visual Studio solution and project files distributed with JEGA (for each of VS .NET 2003, 2005, and 2008).

5.2 Run Time Algorithm Configuration Options

JEGA is the package containing the genetic algorithms that can be run. They are the MOGA and the SOGA. In any given program, JEGA can be used to run multiple instances of MOGAs and SOGAs. Each algorithm that is to be run must be configured individually.

The configuration of a JEGA algorithm is accomplished by the creation and subsequent loading of two configuration objects. The first is an object in which a description of the problem is housed. This object is of the JEGA::FrontEnd::ProblemConfig type. The second is an object in which the details of how JEGA is to operate are housed. This is an instance of the JEGA::FrontEnd::AlgorithmConfig type. These two objects, fully loaded, together with an evaluator and an instance of the JEGA::FrontEnd::Driver class, are all that are necessary to perform optimization using JEGA.

5.2.1 The JEGA Problem Configuration Object

The problem configuration object is used to describe the problem that is to be solved. This includes information such as the number of design variables and what they are like, the number of objective functions and what they are like, and the number of constraints and what they are like. It may be a bit counterintuitive, but the details of the evaluation of the objective functions and constraints are not part of the problem configuration. They are actually part of the algorithm configuration as described below in Section 5.2.2.

Preparing a problem configuration object is as simple as declaring one and loading it. The only real question is what to load (meaning what is available/possible

with JEGA). For the following discussion, consider that a problem configuration object has been declared as:

```
JEGA::FrontEnd::ProblemConfig pConfig;
```

Design Variables

JEGA supports the declaration of both real and integral design variables each with either a continuous or discrete nature. The different types may be mixed at will within a given problem.

Real Variables

Simply stated, real valued variables are those that may have significant digits to the right of the decimal point. If you define a real valued variable with a continuous nature, then you must supply an upper and lower bound for that variable. JEGA will vary the value of that variable to any value within that range (inclusive). In the case of a real valued continuous variable, JEGA also accepts a desired decimal precision value. This value is treated as the number of digits to the right of the decimal point that are of interest. This value may be used by various operators when for example they must convert a real valued variable into a bit string representation.

If you define a real valued variable with a discrete nature, then you must supply JEGA the list of possible values that it can use and it will use only those values.

The following are some examples of informing JEGA of real variables via a problem configuration object.

```
// Adding a simple continuous real variable with bounds [-5, 5]
// and a desired precision of 6 decimal places.
pConfig.AddContinuumRealVariable("X1", -5.0, 5.0, 6);

// Adding a discrete real variable with 4 possible values.
JEGA::DoubleVector disVals;
disVals.push_back(2.7);
disVals.push_back(3.6);
disVals.push_back(1.9);
disVals.push_back(7.35);
pConfig.AddDiscreteRealVariable("X2", disVals);
```

Integral Variables

Integral variables are those that do not have any significant digits to the right of the decimal point. If you define an integral variable with a continuous nature,

then you must supply an upper and lower bound for that variable. JEGA will vary the value of that variable to any integral value within that range (inclusive).

If you define an integral valued variable with a discrete nature, then you must supply JEGA the list of possible values that it can use and it will use only those values.

The following are some examples of informing JEGA of integral variables via a problem configuration object.

```
// Adding a simple continuous integer variable with bounds [-5, 5]
pConfig.AddContinuumIntegerVariable("X3", -5, 5);

// Adding a discrete integer variable with 4 possible values.
JEGA::IntVector disVals;
disVals.push_back(2);
disVals.push_back(5);
disVals.push_back(1);
disVals.push_back(7);
pConfig.AddDiscreteIntegerVariable("X4", disVals);
```

Boolean Variables

Boolean variables are those that can only take on the values of true or false (1 or 0 respectively). The only input required when defining a boolean variable is its label.

Internally, JEGA treats Boolean variables as discrete variables with two possible values. Attempts to add additional values will fail.

The following are some examples of informing JEGA of Boolean variables via a problem configuration object.

```
// Adding a simple Boolean variable
pConfig.AddBooleanVariable("X5");
```

Objective Functions

JEGA supports a number of types of objective function types. These like all problem descriptors used in JEGA are broken up conceptually into a type and a nature. The type of an objective function describes to JEGA how to treat it and how to determine when one value is better than another. The nature in the case of an objective function is one of linear or non-linear. The only time a linear nature should be used is if you are planning to supply coefficients such that JEGA can perform the evaluation. See Section REF!! for an example.

Such evaluations will be a simple weighted sum of design variables. Otherwise, all functions should be declared nonlinear.

Minimize Objectives

This is by far the most common type of objective function. Declaring an objective of type minimize tells JEGA to seek as low a value as possible where $-\infty$ is ideal. If you define a minimize objective with a linear nature, then you must provide a collection of coefficients to multiply the design variables in the evaluation of the objective.

The following are some examples of informing JEGA of minimize objectives via a problem configuration object.

```
// Adding a nonlinear minimization objective.
pConfig.AddNonlinearMinimizeObjective("F1");

// Adding a linear minimization objective.
JEGA::DoubleVector coeffs;
coeffs.push_back(1.7);
coeffs.push_back(7.9);
coeffs.push_back(2.4);
coeffs.push_back(12.5);
pConfig.AddLinearMinimizeObjective("F2", coeffs);
```

In the above example declaration of the linear objective function, the result will be the sum of the first coefficient multiplied by the first design variable, the second coefficient multiplied by the second design variable, etc.

Maximize Objectives

The maximize objective type is the second most common type. Declaring an objective of type maximize tells JEGA to seek as high a value as possible where ∞ is ideal. If you define a maximize objective with a linear nature, then you must provide a collection of coefficients to multiply the design variables in the evaluation of the objective.

The following is an example of informing JEGA of a nonlinear maximize objective via a problem configuration object. To declare a linear maximize objective, perform the same steps as for minimize objectives but replace the *AddLinearMinimizeObjective* function call with *AddLinearMaximizeObjective*.

```
// Adding a nonlinear maximization objective.
pConfig.AddNonlinearMaximizeObjective("F3");
```

Seek Value Objectives

The third objective type available through JEGA is the seek value objective.

Declaring an objective of type seek value tells JEGA to seek as close to a give a value as possible where exact equality with the value is ideal. If you define a seek value objective with a linear nature, then you must provide a collection of coefficients to multiply the design variables in the evaluation of the objective.

The following is an example of informing JEGA of a nonlinear seek value objective via a problem configuration object. To declare a linear seek value objective, perform the same steps as for the previous objective types but replace the *AddLinearObjective* function call with *AddLinearSeekValueObjective*.

```
// Adding a nonlinear seek value objective where 4.7 is the sought value.
pConfig.AddNonlinearSeekValueObjective("F4", 4.7);
```

Seek Range Objectives

The fourth and final objective type available through JEGA is the seek range objective. Declaring an objective of type seek range tells JEGA to seek any values within a given range where any such value is equally ideal and values get worse the further away from that range they get. So for example, if your range were [2, 3], then a 2.4 would be equally as good as a 2.9 and a 1.0 would be equally as bad as a 4.0. If you define a seek range objective with a linear nature, then you must provide a collection of coefficients to multiply the design variables in the evaluation of the objective.

The following is an example of informing JEGA of a nonlinear seek range objective via a problem configuration object. To declare a linear seek range objective, perform the same steps as for the previous objective types but replace the *AddLinearObjective* function call with *AddLinearSeekRangeObjective*.

```
// Adding a nonlinear seek range objective to seek values in
// the range 8.3 to 10.25.
pConfig.AddNonlinearSeekRangeObjective("F4", 8.3, 10.25);
```

Constraints

JEGA supports a number of constraint types. Again, these are broken up conceptually into a type and a nature. The type of a constraint describes to JEGA how to treat it and how to determine whether a value is feasible or not. The nature in the case of a constraint is one of linear or non-linear exactly as it is for objective functions.

Inequality Constraints

This is by far the most common type of constraint. The equation representation of an inequality constraint is:

$$g(\bar{x}) \leq UL \tag{5.1}$$

where UL is some given upper limit, commonly 0. Declaring a constraint of type inequality tells JEGA to seek any value less than or equal to UL where any such value is equally acceptable and any other value is unacceptable. If you define an inequality constraint with a linear nature, then you must provide a collection of coefficients to multiply the design variables in the evaluation of the constraint.

The following are some examples of informing JEGA of inequality constraints via a problem configuration object.

```
// Adding an inequality constraint where the upper limit is 6.2.
pConfig.AddNonlinearInequalityConstraint("G1", 6.2);

// Adding a linear inequality constraint where the upper limit is 0.0.
JEGA::DoubleVector coeffs;
coeffs.push_back(1.7);
coeffs.push_back(7.9);
coeffs.push_back(2.4);
coeffs.push_back(12.5);
pConfig.AddLinearInequalityConstraint("G2", 0.0, coeffs);
```

Two-Sided Inequality Constraints

The equation representation of a two-sided inequality constraint is:

$$LL \leq g(\bar{x}) \leq UL \tag{5.2}$$

where LL is some given lower limit and UL is some given upper limit. Declaring a constraint of type two-sided inequality tells JEGA to seek any value greater than or equal to LL and less than or equal to UL where any such value is equally acceptable and any other value is unacceptable. If you define a two-sided inequality constraint with a linear nature, then you must provide a collection of coefficients to multiply the design variables in the evaluation of the constraint just as for all linear equations.

A two-sided inequality constraint can always be expressed as two simple inequality constraints. This formulation is here for convenience. The following are some examples of informing JEGA of two-sided inequality constraints via a problem configuration object.

```
// Adding a two-sided inequality constraint with lower
// limit of -3.5 and upper limit of 6.2.
pConfig.AddNonlinearTwoSidedInequalityConstraint("G3", -3.5, 6.2);
```

Equality Constraints

The equation representation of an equality constraint in JEGA is:

$$h(\bar{x}) = C \pm \delta \tag{5.3}$$

where C is some constant value (commonly 0) and δ is an allowable violation amount. JEGA uses this value to effectively give the constraint a "thickness". JEGA allows this behavior because equality constraints are notoriously difficult for genetic algorithms to deal with. This value can of course be 0 in which case JEGA will enforce strict equality with C. Declaring an equality constraint in this way tells JEGA to seek any value equal to $C \pm \delta$ where any such value is equally acceptable and any other value is unacceptable. If you define an equality constraint with a linear nature, then you must provide a collection of coefficients to multiply the design variables in the evaluation of the constraint just as for all linear equations.

An equality constraint can always be expressed as two simple inequality constraints. This formulation is here for convenience. The following are some examples of informing JEGA of equality constraints via a problem configuration object.

```
// Adding an equality constraint with target value of 10 and allowable
// violation of 0.05.
pConfig.AddNonlinearEqualityConstraint("G4", 10.0, 0.05);
```

Not-Equality Constraints

The equation representation of a not-equality constraint in JEGA is:

$$h(\bar{x}) \neq C \tag{5.4}$$

where C is some constant value (commonly 0). Declaring a not-equality constraint in this way tells JEGA to seek any value not equal to C where any such value is equally acceptable and the exact value of C is unacceptable. If you define a not-equality constraint with a linear nature, then you must provide a collection of coefficients to multiply the design variables in the evaluation of the constraint just as for all linear equations.

The following are some examples of informing JEGA of not-equality constraints via a problem configuration object.

```
// Adding a not-equality constraint with taboo value of 17.4.
pConfig.AddNonlinearNotEqualityConstraint("G5", 17.4);
```

Once all design variables, constraints, and objective functions are loaded into the problem config, it is complete. The next section begins the discussion of loading an algorithm configuration object.

5.2.2 The JEGA Algorithm Configuration Object

As previously mentioned, the algorithm configuration object is the place in which all information instructing the algorithm on how to behave is stored. It is the loading of the algorithm configuration that requires the most JEGA specific knowledge. In order to inform JEGA what operations to perform, in what order, and what parameter values to use for each, you must be familiar with what is available and how to instruct JEGA in its use.

Figure 5.2 below shows the typical progression of JEGA. Note that since the main loop can be specialized, this is only typical. It is not necessarily the progression of any given run of JEGA and there may be intermediate operations taking place.

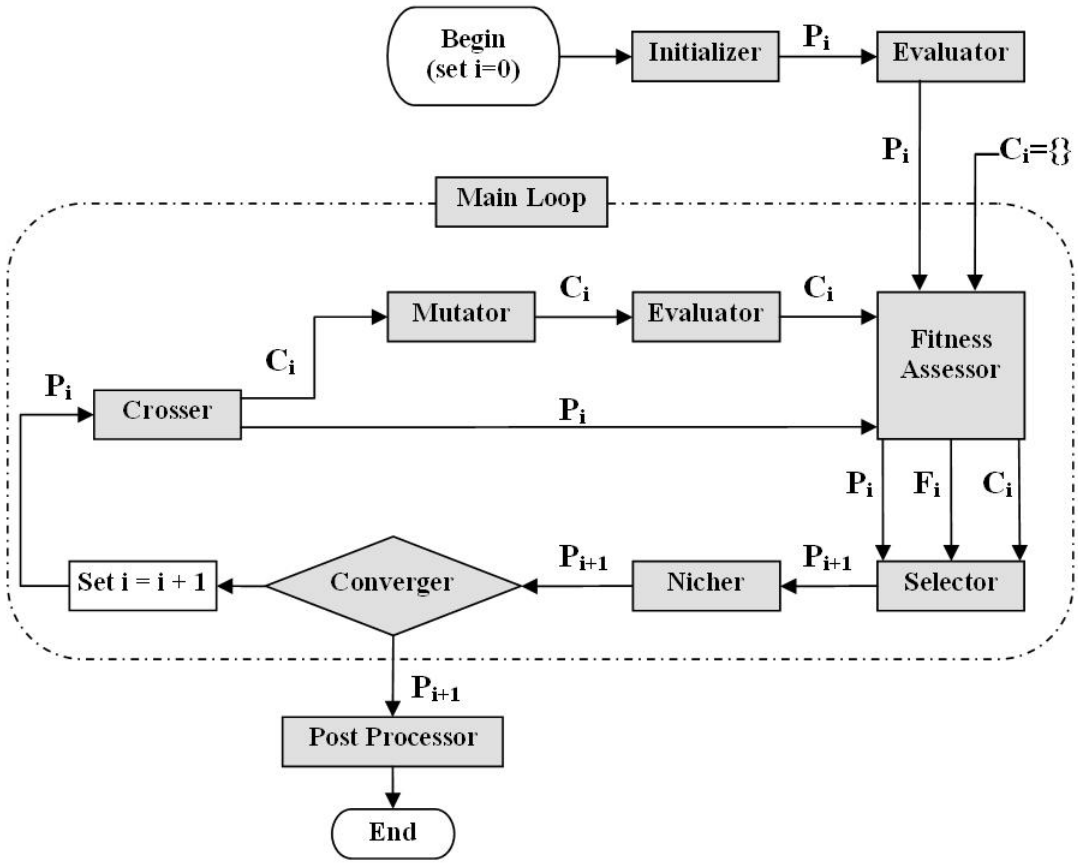

Figure 5.2: The Typical JEGA Algorithm Progression

In order to inform JEGA what versions of each operator to use, you must supply it with a string identifier for each. These strings will be keyed by an-

other string identifier that indicates the class of operator that is being specified. The result is a key-value pair consisting of the operator type identifier followed by the operator specialization identifier. So for example, such a pair may have the form as shown below. Note that the actual input to JEGA will look differently because these arguments will be supplied to methods of the JEGA::FrontEnd::AlgorithmConfig class.

<"initialization_type", "unique_random">

The one exception to this rule is the evaluator which is handled in a different way as described in Section **??**. Table 5.1 below shows each of the operator types along with the strings that identify them, the strings that identify the available specializations common to all algorithms, and the inputs that are common to all specializations. In addition to those in this table, there are operator specializations that are designed specifically for a one or another algorithm. Those will be described later. Note that an input must be supplied for each operator type meaning that they are all required.

Algorithm Independent Inputs

Regardless of which specialization is ultimately chosen for an operator, in many cases there is information required by the base class. This is the information referred to above as being "common to all specializations". That information is also displayed in Table 5.1. Values such as those seen in the table are specified as triplets whereby a key, type, and value are all supplied. An example of such a triplet may have the form shown below. Note again that the actual input to JEGA will look differently.

<"crossover_rate", real, 0.8>

These values must be supplied (or will be given default values) regardless of what specialization is chosen. So for example, no matter what initializer you choose, you must supply a population size or be willing to accept the default of 50.

The inputs of the base class will be used by the specializations in different ways but in general, they have a common meaning. For example, the crossover rate will always be used by a crosser to determine how many crossover operations will take place. However, it may be used differently depending on how the operator works. This will become apparent in Section **??** where the individual operator specializations are described.

In addition to inputs required for operators, the algorithms themselves will require inputs. Again, each algorithm specialization (MOGA or SOGA) may have specialized requirements but each will share the requirements of the base class. The common algorithm inputs are displayed in Table 5.2 below.

The meanings of these inputs are as follows. Remember when reading these descriptions that these inputs pertain to a single algorithm instance within JEGA. A single run of JEGA may involve the creation and marshaling of multiple algorithms.

- **method.algorithm** - Tells JEGA whether to treat the problem as a true multi-objective problem and seek the Pareto optimal solutions or to treat it as a single objective problem and seek the single best solution. Even if you define your problem to have multiple objectives, you can still solve it as a single objective problem by supplying JEGA with weights with which it can combine the multiple objectives into a single objective. This is a required input and has no default value. If not properly supplied, JEGA will cause an exception.

- **method.print_each_pop** - Tells JEGA whether or not to print the current population after all operators in the main loop have been executed at each generation. If this flag is set to true, the populations will be written to files with the pattern "population$< GEN\# >$.dat" where $< GEN\# >$ is the number of the current generation. The files will be written in tab delimited format as follows.

  $dv0 < tab > dv1...dvN[< tab > of0 < tab > of1...ofM < tab > con0 < tab > con1...conK]$

  The objectives and constraints are only written if the design has been evaluated and is not ill-conditioned. The constraints of course are not written if there are none. This is an optional input with a default value of false meaning that the files will not be written.

- **method.jega.algorithm_name** - Tells JEGA what the name of the current algorithm instance is to be. This if of primary use in the creation of logging message. When information is produced for the user by a JEGA algorithm, it is adorned with the name of the algorithm that produced it. This is especially important when JEGA is being used to marshal many algorithms sequentially or in parallel. This is an optional input with a default value that is constructed based on the algorithm type (MOGA or SOGA) and the number of algorithms previously created (ex. MOGA #5).

- **method.output** - Tells JEGA how much output to supply to the user. This input is ignored if logging is not enabled. Each message produced by JEGA is given an importance level. JEGA only outputs those messages whose level is high enough compared to the level defined by this input. The options shown in Table 5.1 are in order of decreasing output. If you choose the "debug" level for example, JEGA will write every single message. If you choose "silent" on the other hand, JEGA will write almost nothing. This is an optional input with a default value of "quiet".

- **method.log_file** - Tells JEGA the name of the file to which you would like the current algorithm to log messages. This input is ignored if file logging is not enabled. This input is optional and by default, the algorithm will log into the global log.

JEGA v2.0 also utilizes the *output* method independent control to vary the amount of information presented to the user during execution.

- Evaluate the new population members.
- Assess the fitness of each member in the population. There are a number of ways to evaluate the fitness of the members of the populations. Choice of fitness assessor operators is strongly dependent on the type of algorithm being used and can have a profound effect on the choice of selectors. For example, if using *MOGA*, the available assessors are the *layer_rank* and *domination_count* fitness assessors. If using either of these, it is strongly recommended that you use the *below_limit* selector as well (although the roulette wheel selectors can also be used). The functionality of the *dominationr_count* selector of JEGA v1.0 can now be achieved using the *domination_count* fitness assessor and *below_limit* selector together. If using *SOGA*, the only fitness assessor is the *merit_function* fitness assessor which currently uses an exterior penalty function formulation to assign fitnesses. Any of the selectors can be used in conjunction with this fitness assessment scheme. This is the first of the selection operators.
- Replace the population with members selected to continue in the next generation. The pool of potential members is the current population and the current set of offspring. The *replacement_type* of *roulette_wheel* or *unique_roulette_wheel* may be used either with MOGA or SOGA problems however they are not recommended for use with MOGA. Given that the only two fitness assessors for MOGA are the *layer_rank* and *domination_count*, the recommended selector is the *below_limit* selector. The *replacement_type* of *favor_feasible* is specific to a SOGA. This replacement operator will always take a feasible design over an infeasible one. Beyond that, it favors solutions based on an assigned fitness value which must have been installed by some fitness assessor.
- Apply niche pressure to the population. This step is specific to the MOGA and is new to JEGA v2.0. Technically, the step is carried out during runs of the SOGA but only the *null_niching* operator is available for use with SOGA. In MOGA, the *radial* niching operator or the *distance* niching operator can be used. The purpose of niching is to encourage differentiation along the Pareto frontier and thus a more even and uniform sampling. In JEGA, niching operators typically work by determining when two designs are too close in the performance (phenotype) space and caching them until the next round of selection.

- Test for convergence. The final step in the iterator loop is to assess the convergence of the algorithm. There are two aspects to convergence that must be considered. The first is stopping criteria. A stopping criteria dictates some sort of limit on the algorithm that is independent of its performance. Examples of stopping criteria available for use with JEGA are the *max_iterations* and *max_function_evaluations* inputs. All JEGA convergers respect these stopping criteria in addition to anything else that they do.

  The second aspect to convergence involves repeated assessment of the algorithms progress in solving the problem. In JEGA v1.0, the fitness tracker convergers (*best_fitness_tracker* and *average_fitness_tracker*) performed this function by asserting that the fitness values (either best or average) of the population continue to improve. There was no such operator for the MOGA. As of JEGA v2.0, the same fitness tracker convergers exist for use with SOGA and there is now a converger available for use with the MOGA. The MOGA converger (*metric_tracker*) operates by tracking various changes in the non-dominated frontier from generation to generation. When the changes occurring over a user specified number of generations fall below a user specified threshold, the algorithm stops.

- Post Process. This occurs after convergence has been attained. This operator provides a means of filtering the data returned by the algorithm in whatever way is relevant. For example, if using a MOGA, it may be desired to return only a certain number of points within a particular region or evenly dispersed throughout the performance space, etc.

There are many controls which can be used for both MOGA and SOGA methods. These include among others the random seed, initialization types, crossover and mutation types, and some replacement types.

The *seed* control defines the starting seed for the random number generator. The algorithm uses random numbers heavily but a specification of a random seed will cause the algorithm to run identically from one trial to the next so long as all other input specifications remain the same. New to JEGA v2.0 is the introduction of the *log_file* specification. JEGA v2.0 uses a logging library to output messages and status to the user. JEGA can be configured at build time to log to both standard error and a text file, one or the other, or neither. The *log_file* input is a string name of a file into which to log. If the build was configured without file logging in JEGA, this input is ignored. If file logging is enabled and no *log_file* is specified, the default file name if JEGAGlobal.log is used. Also new to JEGA v2.0 is the introduction of the *print_each_pop* specification. It serves as a flag and if supplied, the population at each generation will be printed to a file named "population$< GEN\# >$.dat" where $< GEN\# >$ is the number of the current generation.

The *initialization_type* defines the type of initialization for the GA. There are three types: *simple_random*, *unique_random*, and *flat_file*. *simple_random* cre-

ates initial solutions with random variable values according to a uniform random number distribution. It gives no consideration to any previously generated designs. The number of designs is specified by the *population_size*. *unique_random* is the same as *simple_random*, except that when a new solution is generated, it is checked against the rest of the solutions. If it duplicates any of them, it is rejected. *flat_file* allows the initial population to be read from a flat file. If *flat_file* is specified, a file name must be given.

Variables can be delimited in the flat file in any way you see fit with a few exceptions. The delimiter must be the same on any given line of input with the exception of leading and trailing whitespace. So a line could look like: 1.1, 2.2 ,3.3 for example but could not look like: 1.1, 2.2 3.3. The delimiter can vary from line to line within the file which can be useful if data from multiple sources is pasted into the same input file. The delimiter can be any string that does not contain any of the characters .+-dDeE or any of the digits 0-9. The input will be read until the end of the file. The algorithm will discard any configurations for which it was unable to retrieve at least the number of design variables. The objective and constraint entries are not required but if ALL are present, they will be recorded and the design will be tagged as evaluated so that evaluators may choose not to re-evaluate them. Setting the size for this initializer has the effect of requiring a minimum number of designs to create. If this minimum number has not been created once the files are all read, the rest are created using the *unique_random* initializer and then the *simple_random* initializer if necessary.

Note that the *population_size* only sets the size of the initial population. The population size may vary in the JEGA methods according to the type of operators chosen for a particular optimization run.

There are many crossover types available. *multi_point_binary* crossover requires an integer number, N, of crossover points. This crossover type performs a bit switching crossover at N crossover points in the binary encoded genome of two designs. Thus, crossover may occur at any point along a solution chromosome (in the middle of a gene representing a design variable, for example). *multi_point_parameterized_binary* crossover is similar in that it performs a bit switching crossover routine at N crossover points. However, this crossover type performs crossover on each design variable individually. So the individual chromosomes are crossed at N locations. *multi_point_real* crossover performs a variable switching crossover routing at N crossover points in the real real valued genome of two designs. In this scheme, crossover only occurs between design variables (chromosomes). Note that the standard solution chromosome representation in the JEGA algorithm is real encoded and can handle integer or real design variables. For any crossover types that use a binary representation, real variables are converted to long integers by multiplying the real number by 10^6 and then truncating. Note that this assumes a precision of only six decimal places. Discrete variables are represented as integers (indices within a list of possible values) within the algorithm and thus require no special treatment by

the binary operators.

The final crossover type is *shuffle_random*. This crossover type performs crossover by choosing design variables at random from a specified number of parents enough times that the requested number of children are produced. For example, consider the case of 3 parents producing 2 children. This operator would go through and for each design variable, select one of the parents as the donor for the child. So it creates a random shuffle of the parent design variable values. The relative numbers of children and parents are controllable to allow for as much mixing as desired. The more parents involved, the less likely that the children will wind up exact duplicates of the parents.

All crossover types take a *crossover_rate*. The crossover rate is used to calculate the number of crossover operations that take place. The number of crossovers is equal to the rate * *population_size*.

There are five mutation types allowed. *replace_uniform* introduces random variation by first randomly choosing a design variable of a randomly selected design and reassigning it to a random valid value for that variable. No consideration of the current value is given when determining the new value. All mutation types have a *mutation_rate*. The number of mutations for the *replace_uniform* mutator is the product of the *mutation_rate* and the *population_size*.

The *bit_random* mutator introduces random variation by first converting a randomly chosen variable of a randomly chosen design into a binary string. It then flips a randomly chosen bit in the string from a 1 to a 0 or visa versa. In this mutation scheme, the resulting value has more probability of being similar to the original value. The number of mutations performed is the product of the *mutation_rate*, the number of design variables, and the *population_size*.

The offset mutators all act by adding an "offset" random amount to a variable value. The random amount has a mean of zero in all cases. The *offset_normal* mutator introduces random variation by adding a Gaussian random amount to a variable value. The random amount has a standard deviation dependent on the *mutation_scale*. The *mutation_scale* is a fraction in the range [0, 1] and is meant to help control the amount of variation that takes place when a variable is mutated. *mutation_scale* is multiplied by the range of the variable being mutated to serve as standard deviation. *offset_cauchy* is similar to *offset_normal*, except that a Cauchy random variable is added to the variable being mutated. The *mutation_scale* also defines the standard deviation for this mutator. Finally, *offset_uniform* adds a uniform random amount to the variable value. For the *offset_uniform* mutator, the *mutation_scale* is interpreted as a fraction of the total range of the variable. The range of possible deviation amounts is +/- 1/2 * (*mutation_scale* * variable range). The number of mutations for all offset mutators is defined as the product of *mutation_rate* and *population_size*.

As of JEGA v2.0, all replacement types are common to both MOGA and SOGA. They include the *roulette_wheel*, *unique_roulette_wheel*, and *below_limit* selectors. In *roulette_wheel replacement*, each design is conceptually allotted a portion of

a wheel proportional to its fitness relative to the fitnesses of the other Designs. Then, portions of the wheel are chosen at random and the design occupying those portions are duplicated into the next population. Those Designs allotted larger portions of the wheel are more likely to be selected (potentially many times). *unique_roulette_wheel* replacement is the same as *roulette_wheel* replacement, with the exception that a design may only be selected once. The *below_limit* selector attempts to keep all designs for which the negated fitness is below a certain limit. The values are negated to keep with the convention that higher fitness is better. The inputs to the *below_limit* selector are the limit as a real value, and a *shrinkage_percentage* as a real value. The *shrinkage_percentage* defines the minimum amount of selections that will take place if enough designs are available. It is interpreted as a percentage of the population size that must go on to the subsequent generation. To enforce this, *below_limit* makes all the selections it would make anyway and if that is not enough, it takes the remaining that it needs from the best of what is left (effectively raising its limit as far as it must to get the minimum number of selections). It continues until it has made enough selections. The *shrinkage_percentage* is designed to prevent extreme decreases in the population size at any given generation, and thus prevent a big loss of genetic diversity in a very short time. Without a shrinkage limit, a small group of "super" designs may appear and quickly cull the population down to a size on the order of the limiting value. In this case, all the diversity of the population is lost and it is expensive to re-diversify and spread the population.

The specification for controls specific to Multi-objective Evolutionary algorithms are described here. These controls will be appropriate to use if the user has specified *moga* as the method.

The initialization, crossover, and mutation controls were all described in the preceding section. There are no MOGA specific aspects to these controls. The *fitness_type* for a MOGA may be *domination_count* or *layer_rank*. Both have been specifically designed to avoid problems with aggregating and scaling objective function values and transforming them into a single objective. Instead, the *domination_count* fitness assessor works by ordering population members by the negative of the number of designs that dominate them. The values are negated in keeping with the convention that higher fitness is better. The *layer_rank* fitness assessor works by assigning all non-dominated designs a layer of 0, then from what remains, assigning all the non-dominated a layer of -1, and so on until all designs have been assigned a layer. Again, the values are negated for the higher-is-better fitness convention. Use of the *below_limit* selector with the *domination_count* fitness assessor has the effect of keeping all designs that are dominated by fewer then a limiting number of other designs subject to the shrinkage limit. Using it with the *layer_rank* fitness assessor has the effect of keeping all those designs whose layer is below a certain threshold again subject to a minimum.

New to JEGA v2.0 is the introduction of niche pressure operators. These operators are meant primarily for use with the moga. The job of a niche pressure

operator is to encourage diversity along the Pareto frontier as the algorithm runs. This is typically accomplished by discouraging clustering of design points in the performance space. In JEGA, the application of niche pressure occurs as a secondary selection operation. The nicher is given a chance to perform a pre-selection operation prior to the operation of the selection (replacement) operator, and is then called to perform niching on the set of designs that were selected by the selection operator.

Currently, the only niche pressure operator available is the *radial* nicher. This niche pressure applicator works by enforcing a minimum distance between designs in the performance space at each generation. The algorithm proceeds by starting at the (or one of the) extreme designs along objective dimension 0 and marching through the population removing all designs that are too close to the current design. One exception to the rule is that the algorithm will never remove an extreme design which is defined as a design that is maximal or minimal in all but 1 objective dimension (for a classical 2 objective problem, the extreme designs are those at the tips of the non-dominated frontier).

The designs that are removed by the nicher are not discarded. They are buffered and re-inserted into the population during the next pre-selection operation. This way, the selector is still the only operator that discards designs and the algorithm will not waste time "re-filling" gaps created by the nicher.

The niche pressure control consists of two options. The first is the *null_niching* option which specifies that no niche pressure is to be applied. The second is the *radial_niching* option which specifies that the radial niching algorithm is to be use. The radial nicher requires as input a vector of fractions with length equal to the number of objectives. The elements of the vector are interpreted as percentages of the non-dominated range for each objective defining a minimum distance to all other designs. All values should be in the range (0, 1). The minimum allowable distance between any two designs in the performance space is the euclidian distance defined by these percentages.

Also new to JEGA v2.0 is the introduction of the MOGA specific *metric_tracker* converger. This converger is conceptually similar to the best and average fitness tracker convergers in that it tracks the progress of the population over a certain number of generations and stops when the progress falls below a certain threshold. The implementation is quite different however. The *metric_tracker* converger tracks 3 metrics specific to the non-dominated frontier from generation to generation. All 3 of these metrics are computed as percent changes between the generations. In order to compute these metrics, the converger stores a duplicate of the non-dominated frontier at each generation for comparison to the non-dominated frontier of the next generation.

The first metric is one that indicates how the expanse of the frontier is changing. The expanse along a given objective is defined by the range of values existing within the non-dominated set. The expansion metric is computed by tracking the extremes of the non-dominated frontier from one generation to the next.

Any movement of the extreme values is noticed and the maximum percentage movement is computed as:

$$E_m = \max_{j=1}^{nof} \left| \frac{range(j,i) - range(j,i-1)}{range(j,i-1)} \right| \tag{5.5}$$

where E_m is the max expansion metric, j is the objective function index, i is the current generation number, and nof is the total number of objectives. The range is the difference between the largest value along an objective and the smallest when considering only non-dominated designs.

The second metric monitors changes in the density of the non-dominated set. The density metric is computed as the number of non-dominated points divided by the hypervolume of the non-dominated region of space. Therefore, changes in the density can be caused by changes in the number of non-dominated points or by changes in size of the non-dominated space or both. The size of the non-dominated space is computed as:

$$V_{ps}(i) = \prod_{j=1}^{nof} range(j,i) \tag{5.6}$$

where $V_{ps}(i)$ is the hypervolume of the non-dominated space at generation i and all other terms have the same meanings as above.

The density of the a given non-dominated space is then:

$$D_{ps}(i) = \frac{P_{ct}(i)}{V_{ps}(i)} \tag{5.7}$$

where $P_{ct}(i)$ is the number of points on the non-dominated frontier at generation i.

The percentage increase in density of the frontier is then calculated as:

$$C_d = \left| \frac{D_{ps}(i) - D_{ps}(i-1)}{D_{ps}(i-1)} \right| \tag{5.8}$$

where C_d is the change in density metric.

The final metric is one that monitors the "goodness" of the non-dominated frontier. This metric is computed by considering each design in the previous population and determining if it is dominated by any designs in the current population. All that are determined to be dominated are counted. The metric is the ratio of the number that are dominated to the total number that exist in the previous population.

As mentioned above, each of these metrics is a percentage. The tracker records the largest of these three at each generation. Once the recorded percentage is below the supplied percent change for the supplied number of generations consecutively, the algorithm is converged.

The specification for convergence in a MOGA can either be *metric_tracker* or can be omitted all together. If omitted, no convergence algorithm will be used and the algorithm will rely on stopping criteria only. If *metric_tracker* is specified, then a *percent_change* and *num_generations* must be supplied as with the other metric tracker convergers (average and best fitness trackers). The *percent_change* is the threshold beneath which convergence is attained whereby it is compared to the metric value computed as described above. The *num_generations* is the number of generations over which the metric value should be tracked. Convergence will be attained if the recorded metric is below *percent_change* for *num_generations* consecutive generations.

The MOGA specific controls are described in below. Note that MOGA and SOGA create additional output files during execution. "finaldata.dat" is a file that holds the Pareto members of the population in the final generation. "discards.dat" holds solutions that were discarded from the population during the course of evolution. It can often be useful to plot objective function values from these files to visually see the Pareto front and ensure that finaldata.dat solutions dominate discards.dat solutions. The solutions are written to these output files in the format "Input1...InputN..Output1...OutputM". If MOGA is used in a multi-level optimization strategy (which requires one optimal solution from each individual optimization method to be passed to the subsequent optimization method as its starting point), the solution in the Pareto set closest to the "utopia" point is given as the best solution. This solution is also reported in the DAKOTA output. This "best" solution in the Pareto set has minimum distance from the utopia point. The utopia point is defined as the point of extreme (best) values for each objective function. For example, if the Pareto front is bounded by (1,100) and (90,2), then (1,2) is the utopia point. There will be a point in the Pareto set that has minimum L2-norm distance to this point, for example (10,10) may be such a point. In SOGA, the solution that minimizes the single objective function is returned as the best solution.

The specification for controls specific to Single-objective Evolutionary algorithms are described here. These controls will be appropriate to use if the user has specified *soga* as the method.

The initialization, crossover, and mutation controls were all described above. There are no SOGA specific aspects to these controls. The *replacement_type* for a SOGA may be *roulette_wheel*, *unique_roulette_wheel*, or *favor_feasible*. The *favor_feasible* replacement type always takes a feasible design over an infeasible one. Beyond that, it selects designs based on a fitness value. As of JEGA v2.0, the fitness assessment operator must be specified with SOGA although the *merit_function* is currently the only one. The roulette wheel selectors no longer assume a fitness function. The *merit_function* fitness assessor uses an exterior

penalty function formulation to penalize infeasible designs. The specification allows the input of a *constraint_penalty* which is the multiplier to use on the constraint violations.

The SOGA controls allow two additional convergence types. The *convergence_type* called *average_fitness_tracker* keeps track of the average fitness in a population. If this average fitness does not change more than *percent_change* over some number of generations, *num_generations*, then the solution is reported as converged and the algorithm terminates. The *best_fitness_tracker* works in a similar manner, only it tracks the best fitness in the population. Convergence occurs after *num_generations* has passed and there has been less than *percent_change* in the best fitness value. Both also respect the stopping criteria.

The SOGA specific controls are described in below.

Operator	Identifier	Type	Options	Status	Default
Main Loops	method.jega.mainloop_type	String	duplicate_free null_main_loop standard	Required	
Initializers	method.initialization_type	String	double_matrix flat_file null_initialization random unique_random	Required	
	method.population_size	Integer	[0, ∞]	Optional	50
Mutators	method.mutation_type	String	bit_random null_mutation offset_cauchy offset_normal offset_uniform replace_uniform	Required	
	method.mutation_rate	Real	[0.0, 1.0]	Optional	0.05
Crossers	method.crossover_type	String	multi_point_binary multi_point_parameterized_binary multi_point_real null_crossover shuffle_random	Required	
	method.crossover_rate	Real	[0.0, 1.0]	Optional	0.75
Convergers	method.jega.convergence_type	String	average_fitness_tracker best_fitness_tracker max_evals_or_gens max_evaluations max_generations null_convergence	Required	
	method.max_iterations	Integer	[0, ∞]	Optional	∞
	method.max_function_evaluations	Integer	[0, ∞]	Optional	∞
Evaluators			null_evaluation		
	method.max_function_evaluations	Integer	[0, ∞]	Optional	∞
Fitness Assessors	method.jega.fitness_type	String	null_fitness	Required	
Selectors	method.replacement_type	String	below_limit null_selection roulette_wheel unique_roulette_wheel	Required	
Nichers	method.jega.niching_type	String	null_niching	Required	
	method.jega.cache_niched_designs	Boolean	true or false	Optional	true
Post Processors	method.jega.postprocessor_type	String	null_postprocessor distance_postprocessor	Required	

Table 5.1: Operator Class Input Requirements.

Identifier	Type	Options	Status	Default
method.algorithm	String	moga soga	Required	
method.print_each_pop	Boolean	true or false	Optional	false
method.jega.algorithm_name	String		Optional	<type> #<instance #>
method.output	Integral	debug (10) verbose (20) quiet (30) silent (40) fatal (50)	Optional	quiet (30)
method.log_file	String		Optional	JEGAGlobal.log

Table 5.2: Algorithm Class Input Requirements.

Chapter 6

Capabilities

Chapter 7

Output

Chapter 8

Examples

This chapter provides some example applications for running JEGA. This is not meant to show off the capabilities of JEGA but instead demonstrate how to use it. Therefore, all examples in this chapter will solve the same test problem.

The test problem is a case where the Pareto frontier is continuous and concave. The problem is to simultaneously optimize f_1 and f_2 given three input variables, x_1, x_2, and x_3, where the inputs are bounded by $-4 \leq x_i \leq 4$:

$$f_1(x) = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i - \frac{1}{\sqrt{3}}\right)^2\right)$$
$$f_2(x) = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i + \frac{1}{\sqrt{3}}\right)^2\right)$$

A solution to this problem generated using JEGA can be seen in Figure 8.1 below.

8.1 Example 1 - A Minimal Implementation

This section displays the minimal amount of code necessary to solve this problem using JEGA. It is not a "best-practice" implementation. To achieve the minimal implementation, this example makes use of the core SimpleFunctorEvaluator and the associated front end SimpleFunctorEvaluatorCreator.

As with any C++ program, we will need a main function. As described in Chapter 5, we will also need a problem configuration object, an algorithm configuration object, an evaluator and associated creator, and a parameter database.

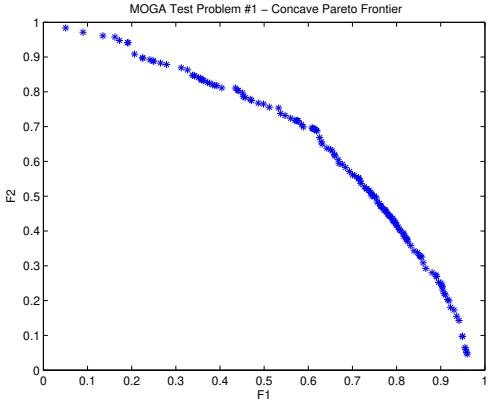

Figure 8.1: The Pareto Frontier of the Example Problem.

```
#include <cmath>
#include <memory>
#include <iostream>
#include <../FrontEnd/core/include/Driver.hpp>
#include <../FrontEnd/core/include/ProblemConfig.hpp>
#include <../FrontEnd/core/include/AlgorithmConfig.hpp>
#include <../Utilities/include/BasicParameterDatabaseImpl.hpp>
#include <../FrontEnd/core/include/SimpleFunctorEvaluatorCreator.hpp>

struct MyEvaluationFunctor :
    public JEGA::Algorithms::SimpleFunctorEvaluator::Functor
{
        virtual bool Evaluate(
            const JEGA::DoubleVector& X,
            JEGA::DoubleVector& F,
            JEGA::DoubleVector& G
            )
        {
            const std::size_t ndv = X.size();
            static const double sqrt3inv = 1.0/std::sqrt(3.0);

            F[0] = F[1] = 0.0;

            for(size_t dv=0; dv<ndv; ++dv) {
                F[0] += std::pow(X[dv] - (sqrt3inv), 2.0);
                F[1] += std::pow(X[dv] + (sqrt3inv), 2.0);
            }
```

```
            F[0] = 1.0 - std::exp(-F[0]);
            F[1] = 1.0 - std::exp(-F[1]);

            return true;
        }
};

int main(int argc, char* argv[])
{
    // All programs must initialize JEGA once and only once.
    JEGA::FrontEnd::Driver::InitializeJEGA(
        "JEGAGlobal.log", JEGA::Logging::ldebug(), 0
        );

    // Load up a problem config.
    JEGA::FrontEnd::ProblemConfig pConfig;
    pConfig.AddContinuumRealVariable("x1", -4.0, 4.0, 6);
    pConfig.AddContinuumRealVariable("x2", -4.0, 4.0, 6);
    pConfig.AddContinuumRealVariable("x3", -4.0, 4.0, 6);
    pConfig.AddNonlinearMinimizeObjective("F1");
    pConfig.AddNonlinearMinimizeObjective("F2");

    // Now load up an algorithm config for which we'll need a parameter
    // database and an evaluator creator.
    JEGA::Utilities::BasicParameterDatabaseImpl pdb;
    std::auto_ptr<MyEvaluationFunctor> ef(new MyEvaluationFunctor());
    JEGA::FrontEnd::SimpleFunctorEvaluatorCreator ec(ef.get());
    JEGA::FrontEnd::AlgorithmConfig aConfig(ec, pdb);

    // Start with algorithm level configuration.
    aConfig.SetAlgorithmType(JEGA::FrontEnd::AlgorithmConfig::MOGA);
    aConfig.SetDefaultLoggingLevel(JEGA::Logging::ldebug());
    aConfig.SetAlgorithmName("MOGA_1");
    aConfig.SetPrintPopEachGen(false);
    aConfig.SetOutputFilenamePattern("finaldata#.dat");

    // Now move on to operator configurations.
    aConfig.SetConvergerName("metric_tracker");
    pdb.AddIntegralParam("method.max_iterations", 2147483647);
    pdb.AddIntegralParam("method.max_function_evaluations", 3000);
    pdb.AddDoubleParam("method.jega.percent_change", 0.03);
    pdb.AddSizeTypeParam("method.jega.num_generations", 10);

    aConfig.SetCrosserName("multi_point_binary");
    pdb.AddDoubleParam("method.crossover_rate", 0.8);
    pdb.AddSizeTypeParam("method.jega.num_cross_points", 2);
```

```
    aConfig.SetNichePressureApplicatorName("distance");
    pdb.AddDoubleVectorParam(
        "method.jega.niche_vector", JEGA::DoubleVector(2, 0.05)
        );
    pdb.AddBooleanParam("method.jega.cache_niched_designs", true);

    aConfig.SetFitnessAssessorName("domination_count");

    aConfig.SetInitializerName("unique_random");
    pdb.AddIntegralParam("method.population_size", 50);

    aConfig.SetMainLoopName("duplicate_free");

    aConfig.SetMutatorName("replace_uniform");
    pdb.AddDoubleParam("method.mutation_rate", 0.1);

    aConfig.SetSelectorName("below_limit");
    pdb.AddDoubleParam("method.jega.fitness_limit", 4);
    pdb.AddDoubleParam("method.jega.shrinkage_percentage", 0.9);

    aConfig.SetPostProcessorName("null_postprocessor");

    // Now instantiate and use a Driver to get the
    // solutions to the problem.
    JEGA::FrontEnd::Driver app(pConfig);
    JEGA::Utilities::DesignOFSortSet res(
        app.ExecuteAlgorithm(aConfig)
        );

    // Do something with the solutions here.  We'll just print them
    // to the console for now.
    res.stream_out(std::cerr) << "\n\n";

    // YOU MUST FLUSH THE RETURNED SET OF SOLUTIONS
    // TO AVOID A MEMORY LEAK!!
    res.flush();
}
```

Bibliography