SANDIA REPORT

SAND2009-6265
Unlimited Release
Printed October 2009

HOPSPACK 2.0 User Manual (v 2.0.2)

Todd D. Plantenga

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.aspx

SAND2009-6265
Unlimited Release
Printed October 2009

HOPSPACK 2.0 User Manual (v 2.0.2)

Todd D. Plantenga
Informatics and Decision Science Department
Sandia National Laboratories
Livermore, CA 94551-9159
Email: tplante@sandia.gov

Abstract

HOPSPACK (Hybrid Optimization Parallel Search PACKage) solves derivative-free optimization problems using an open source, C++ software framework. The framework enables parallel operation using MPI or multithreading, and allows multiple solvers to run simultaneously and interact to find solution points. HOPSPACK comes with an asynchronous pattern search solver that handles general optimization problems with linear and nonlinear constraints, and continuous and integer-valued variables. This user manual explains how to install and use HOPSPACK to solve problems, and how to create custom solvers within the framework.

This SAND report was first issued in October 2009 as the User Manual for HOPSPACK 2.0. Minor revisions to the manual were made for subsequent minor releases of the software.

User Manual revision history

2.0	Oct 2009	First HOPSPACK User Manual.
2.0.1	Mar 2010	Added instructions for building on Mac OSX (Section 6.4), and clarified the return status when evaluating by System Call (Section 5.1).
2.0.2	Apr 2011	Added an example of linking Fortran LAPACK libraries (Section 6.3), and added information about scaling of variables (Section 4).

Acknowledgments

This work was funded by the U.S. Department of Energy, through the Office of Advanced Scientific Computing Research (ASCR), as part of the Applied Mathematics Research Program (http://www.er.doe.gov/ascr/Research/AppliedMath.html).

The developers thank Joshua Griffin for his preliminary work on "HOPSPACK 1.0", which tested many ideas and formed the basis of the current software.

Thanks to Professor Komei Fukuda for allowing the use of CDDLIB source code in the GSS solver.

Contents

Figures

This page intentionally left blank.

1 Introduction

HOPSPACK (Hybrid Optimization Parallel Search PACKage) is derivative-free optimization software for solving general optimization problems, especially those with noisy and expensive functions. HOPSPACK provides an open source C++ framework that enables parallel operation using MPI (for distributed processing architectures) or multithreading (for multi-core machines). The software is easily interfaced with application code, builds on most operating systems (Linux, Windows, Mac OSX), and is designed for extension and customization.

The basic optimization problem addressed is

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & A_{\mathcal{I}}\, x \geq b_{\mathcal{I}} \\ & A_{\mathcal{E}}\, x = b_{\mathcal{E}} \\ & c_{\mathcal{I}}(x) \geq 0 \\ & c_{\mathcal{E}}(x) = 0 \\ & l \leq x \leq u \end{array} \tag{1}$$

Here $f(x) : \mathcal{R}^n \rightarrow \mathcal{R} \cup \{+\infty\}$ is the objective function of n unknowns. The first two constraints specify linear inequalities and equalities with coefficient matrices $A_{\mathcal{I}}$ and $A_{\mathcal{E}}$. The next two constraints describe nonlinear inequalities and equalities captured in functions $c_{\mathcal{I}}(x)$ and $c_{\mathcal{E}}(x)$. The final constraints denote lower and upper bounds on the variables. HOPSPACK allow variables to be continuous or integer-valued and has provisions for multi-objective optimization problems. In general, functions $f(x), c_{\mathcal{I}}(x),$ and $c_{\mathcal{E}}(x)$ can be noisy and nonsmooth, although most algorithms perform best on determinate functions with continuous derivatives.

HOPSPACK is released with two user communities in mind: those who need an optimization problem solved, and those who wish to experiment with writing their own derivative-free solvers.

Key features for users who need to solve an optimization problem are:

- **Only function values are required for the optimization** (no derivatives), so it can be applied to a wide variety of problems. Functions can be nonsmooth or noisy, and take any amount of time to execute (for example, complex simulations may take minutes or hours to run).
- **The user simply provides a program** that can evaluate the objective and nonlinear constraint functions at a given point. The program can be written in any language: Fortran, C/C++, Perl, MATLAB, etc. The procedure for evaluating the objective and constraint functions does not require an encapsulating subroutine wrapper or linking with HOPSPACK libraries; usually the procedure is an entirely separate program.
- **HOPSPACK can be run in parallel** on a cluster of computers or on multi-core machines, greatly reducing the total solution time. Parallelism is achieved by assigning function evaluations of individual points to different processors, which automatically gives good load balancing. Asynchronous operation can use either MPI for distributed machine parallelism, or multithreading for parallel operation on multi-core machines.
- **An asynchronous implementation of the Generating Set Search (GSS) algorithm is supplied.** GSS is a type of pattern search solver that was available in the predecessor to

HOPSPACK. The core GSS solver handles linear constraints, and is extended in HOPSPACK 2.0 to allow nonlinear constraints, integer-valued variables, and multiple start points.

- **Multiple algorithms can run simultaneously** and are easily configured to share information, leading to a faster "best" solution.
- **Binary executables are available for single machine systems.** Executables are multithreaded to utilize multiple processors or cores on the machine.
- **Source code builds with native C++ compilers on Linux, Mac OSX, and Windows.** The source is easily configured to compile with MPI implementations and third-party libraries.

Key features for users developing their own derivative-free solvers are:

- **Parallel evaluation of trial points is managed** by the HOPSPACK framework, exploiting both distributed machine parallelism and multithreading.
- **A simple C++ interface cleanly abstracts framework utilities** and the application's problem definition. An algorithm iterates by receiving a list of newly evaluated points, and then submitting a list of unevaluated trial points. All other work is handled by the HOPSPACK framework.
- **Algorithms share a cache of computed function and constraint evaluations** to eliminate duplicate work.
- **Algorithms can initiate and control subproblems**, a useful technique for handling multiple start points, nonlinear constraints, and integer-valued variables.
- **Source code ports easily to compilers on most operating systems**, including GNU gcc, Intel C++, and Microsoft Visual Studio C++.
- The software is freely available under the terms of the GNU Lesser General Public License.

1.1 Project History

HOPSPACK is a successor to Sandia National Laboratory's APPSPACK (Asynchronous Parallel Pattern Search PACKage) product. The final version of APPSPACK, 5.0, was released in 2007 [1, 4]. The HOPSPACK software builds on APPSPACK 5.0, extending its capabilities in several ways:

- **Nonlinear constraints and integer-valued variables** are accepted by the framework and handled by extensions to the GSS solver.
- **Multithreading** capability is provided on a machine with multiple processors or cores. This allows parallel processing without installing MPI and compiling source code.
- **Multiple solvers** can run simultaneously and share information.

- **Solvers can initiate and control subproblem solvers**.
- **Windows and Mac OSX native compilers** are supported.

Both APPSPACK and HOPSPACK projects were led by Tamara G. Kolda of Sandia National Laboratories (SNL). Major contributors to APPSPACK were Genetha Gray (SNL), Joshua Griffin (SAS Institute), Patty Hough (SNL), Michael Lewis (William & Mary), and Virginia Torczon (William & Mary).

Both projects make use of source code from CDDLIB 0.94 in the GSS solver, generously made available by permission of Professor Komei Fukuda (http://www.ifor.math.ethz.ch/staff/fukuda).

An early version of HOPSPACK was developed and coded by Tamara Kolda and Joshua Griffin, who was then with Sandia National Laboratories. Work on HOPSPACK 2.0 was completed in 2009 by Todd Plantenga.

1.2 Citing HOPSPACK

If you find HOPSPACK useful, please cite this technical report in any resulting publications or reports. The BibTex reference is as follows:

```
@TECHREPORT{Hops20-Sandia,
  author = {Todd D. Plantenga},
  title  = {HOPSPACK 2.0 User Manual},
  institution = {Sandia National Laboratories, Albuquerque, NM and Livermore, CA},
  month  = {October},
  year   = {2009},
  number = {SAND2009-6265}
}
```

We greatly appreciate hearing about applications and success stories using HOPSPACK. Such information helps determine the next improvements to be made in the software, and helps us to continue funding this work. Please email the author at **tplante@sandia.gov**, or visit the Wiki page at https://software.sandia.gov/trac/hopspack.

2 Quick Start

The fastest way to start using HOPSPACK is to download a precompiled executable package and interface your optimization problem based on the examples provided with the package. The precompiled code is limited to a single machine, but can parallel process using threads on a machine with multiple processors or cores. Precompiled executables do not require additional third party software installs. Packages are available for:

Linux. 32-bit x86 processors, compiled with g++ 3.4.6 on Red Hat Enterprise Linux WS 4.

Mac OSX. 32-bit x86 processors, compiled with g++ 4.0.1 (XCode 3.1.2) on Mac OSX 10.5.8.

Windows. 32-bit x86 processors, compiled with Microsoft Visual C++ 9.0 on Windows XP (SP2). Should run on 32-bit Windows Vista, Windows Server 2003, and Windows Server 2008. Should also run in 32-bit emulation mode (WOW64) on Windows XP Professional x64.

Follow these steps to quickly solve your optimization problem:

- **Download a package for your machine.** Follow the links at
  https://software.sandia.gov/trac/hopspack

  Get a binary package, save it in any directory, and unzip the file. No administrative privileges are needed.

- **Run an example.** Open a command line terminal window. Find the directory where you unzipped HOPSPACK and change to examples/1-var-bnds/only (on Windows use backslashes '\' instead of '/'). Now type

  ```
  > ../../HOPSPACK_main_serial example1_params.txt
  ```

  Compare the answer with results in the file examples/README.txt.

- **Learn how to configure a parameter file.** For instance, to get a more accurate solution to the first example, edit the text file example1_params.txt and change the `Step Tolerance` in the "Citizen 1" sublist from 0.01 to 0.002. Then run the example again. Read Section 4 to learn about configuration parameters, especially the example parameter file at the beginning of the section.

- **Interface your problem with HOPSPACK.** Create your own parameter file, based on one of the examples. The number of variables, bounds, and linear constraints are specified in this file (details are in Section 4.3 and Section 4.4). Write a simple script or program that computes the objective function and any nonlinear constraint values (for instance, see examples/2-linear-constraints/linear_constraints.cpp). The program should take an input file name, evaluate the functions, and write the answer to a file. Edit the parameter file and put the name of your program as the `Executable Name` in sublist "Evaluator". Read Section 4.1 to learn more about formulating your optimization problem, and Section 5 to learn more about the evaluation step.

- **Run your problem.** Invoke HOPSPACK with your parameter file name:

```
> ../../HOPSPACK_main_serial your_params.txt
```

If your machine has multiple processors or cores, then you can try parallel evaluations:

```
> ../../HOPSPACK_main_threaded your_params.txt
```

Edit the parameter file and increase the `Number Threads` parameter to engage more CPU resources.

What to do next:

Build HOPSPACK. The precompiled code is limited to multithreaded parallelization on one machine, and executes linear algebra routines with the standard Netlib LAPACK library. You can download source code and build with your own compiler, compile with MPI for distributed machine operation, link with a different LAPACK library, and more. Read Section 6 to learn how.

Extend HOPSPACK. You can download source code and make modifications to suit special needs. For example, you can embed HOPSPACK in other software, call the application that computes function values directly, or change the way parallel resources are allocated. Read Section 7 to learn more.

Write your own solver. This is what the HOPSPACK framework is intended for. You can download source code and add your own algorithm in a new solver. You might write a global search solver that controls the GSS local solver in HOPSPACK, or write your own local search solver, or hybridize different algorithms. Start with Section 3.1 to learn about the framework, read the description of the GSS solver in Section 3.3, and refer to Section 7 to learn about extending the software.

3 Theory of Operation

The primary goal of HOPSPACK is to provide a parallel processing framework for executing algorithms that solve optimization problems with no derivatives (please note that the terms "solver" and "algorithm" are used interchangeably in this document). This section describes the architecture of the software framework and the suite of GSS solvers included with HOPSPACK. You should study this section carefully before writing your own algorithm as a HOPSPACK solver.

3.1 Software Architecture

In HOPSPACK each solver is called a Citizen. Citizens are independent, but share the resources of the HOPSPACK framework. Different types of algorithms are coded as different subclasses of a Citizen base class, which provides uniform access to the framework.

Figure 1 shows the major components of the HOPSPACK framework. The large box in the center contains the single "main thread" (it could be a thread or a process) that runs Citizens, the Mediator, and the Conveyor (these components are described below). Workers of two different types execute in parallel with the main thread. On the right are workers for evaluating functions at a particular trial point, and on the left are workers associated with specific citizens.

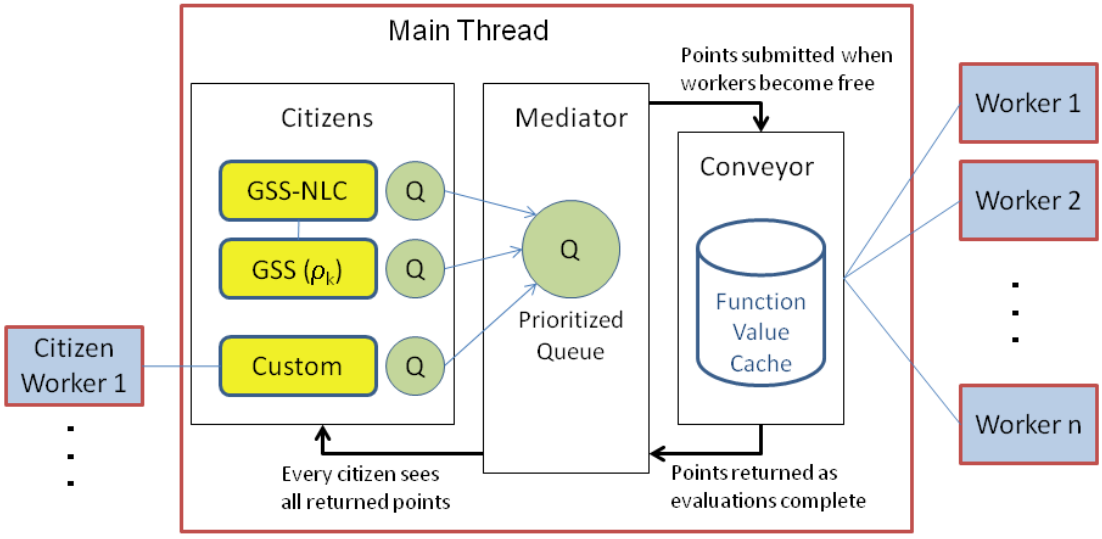


Figure 1. HOPSPACK architecture diagram.

The Mediator runs a main processing loop until deciding that HOPSPACK should stop. Each Mediator iteration assembles trial point requests from all Citizens and passes them to the Conveyor. The Conveyor checks if new points can be fulfilled from a cache, and sends the rest to idle workers for evaluation. Two caches are checked: a primary Cache of completed evaluations, and a Pending Cache that lists trial points currently assigned to evaluation workers. The Conveyor collects results from workers that have completed and passes these back to the Mediator. Citizens are then given the full set of newly evaluated points. They examine values and submit new trial points, which starts the next iteration. Citizens can also spawn dynamic "child citizens" to solve subproblems.

From a citizen's point of view trial points are evaluated asynchronously, so requests are typically managed with an internal queue, as shown in Figure 1. The citizen submits trial points and may receive evaluated results at any future iteration. Evaluated points follow the order given from the citizen's queue of trial points. The citizen has the opportunity to retract previously submitted points if they are still waiting on the Conveyor (for example, the GSS citizen will retract old unevaluated requests when a new "best point" is found). The Mediator collects trial requests from each citizen into a single queue, interleaving points based on citizen priorities.

Figure 1 shows three citizen instances running simultaneously. The top two are connected because the GSS-NLC instance dynamically created the GSS instance to solve a subproblem. The citizen labeled DIRECT has a parallel processing worker because its algorithm requires significant CPU time to process points. The "citizen worker" allows the Mediator loop to run more quickly, and thus avoids slowing down other citizens.

As coordinator of the "town" of citizens, the Mediator decides when to stop HOPSPACK and what final solution to return. Each citizen decides when it has converged to a solution point, as defined by its own criteria. If all citizens have converged not necessarily to the same point), then the Mediator stops. It reports the best feasible point that it has seen, regardless of which citizen generated the point. If a problem-specific stop rule is supplied (for example, an `Objective Target` value, see p 25), then the Mediator will stop as soon as it sees a point satisfying the criteria (see Section 3.2). The Mediator will also stop execution if a defined resource limit is reached (for instance, execution time or the total number of worker evaluations).

The Cache in Figure 1 is an important feature of HOPSPACK that often improves performance of GSS and related algorithms. The Cache remembers all points and values that have been evaluated. If a new trial point is sufficiently close to a cached value, then it reports the stored value instead of making another evaluation. The definition of "closeness" is controlled by the `Cache Comparison Tolerance` parameter (p 33). The Cache can also write its contents to a file (`Cache Output File`, p 34), and load from a file (`Cache Input File`, p 33) when HOPSPACK initializes. This feature allows HOPSPACK to be interrupted without losing work. As an interesting exercise, try solving a problem and saving the cache to a file, and then solving again after initializing from the cache file. If the Mediator is instructed to use `Synchronous Evaluations` (p 34), then the problem will solve completely from cached information. If not synchronous then there may be a few new iterations that choose different directions, but the citizen should quickly build up a cache that can solve without any evaluations.

Workers run copies of the application to collect objective and nonlinear constraint values at trial points. Each worker runs a HOPSPACK Evaluator instance, which typically calls the application as a separate process for each trial point. See Section 5 for details on how an application is called. The workers on the right side of Figure 1 run in parallel under direction of the Conveyor. The Conveyor uses an Executor subclass, specialized either for MPI or multithreaded (MT) operation, to coordinate workers. Figure 2 shows an example of the worker partitioning for HOPSPACK using MPI on three nodes, and Figure 3 shows an example of worker partitioning for HOPSPACK using multithreading on a quad-core machine.

MPI and MT are complementary methods of obtaining parallel performance. MPI-based HOPSPACK can solve problems on computing clusters with tens of thousands of nodes, while MT exploits the multi-core capacity of a single machine. MT binaries are available for most platforms, but MPI requires recompiling HOPSPACK source code with an MPI-aware compiler.

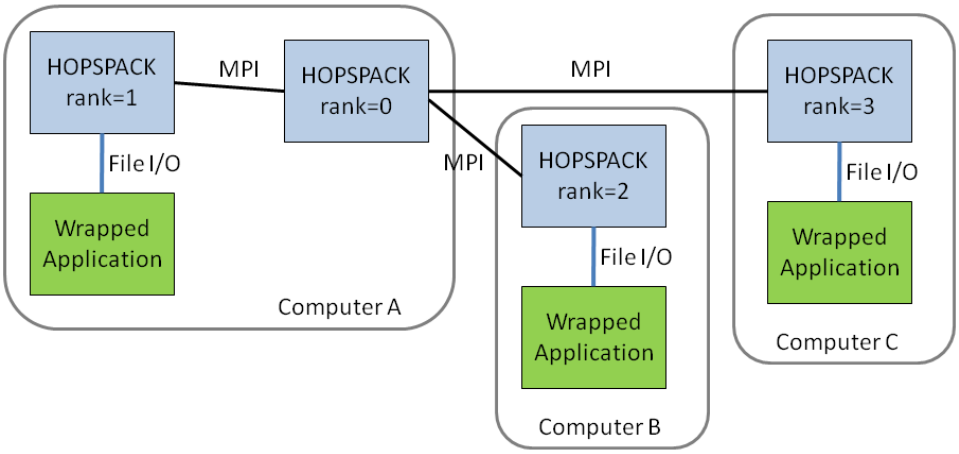


Figure 2. HOPSPACK communication using MPI.

In this example three copies of the application run in parallel on three different computers. Four MPI nodes are employed. Evaluator nodes (rank 1, 2, 3) make system calls to the application, and the main node (rank 0) runs the HOPSPACK Executor to control evaluations. The Mediator and Citizen solvers also run on the main node. Execution may be faster if the main node can be placed on a separate computer.

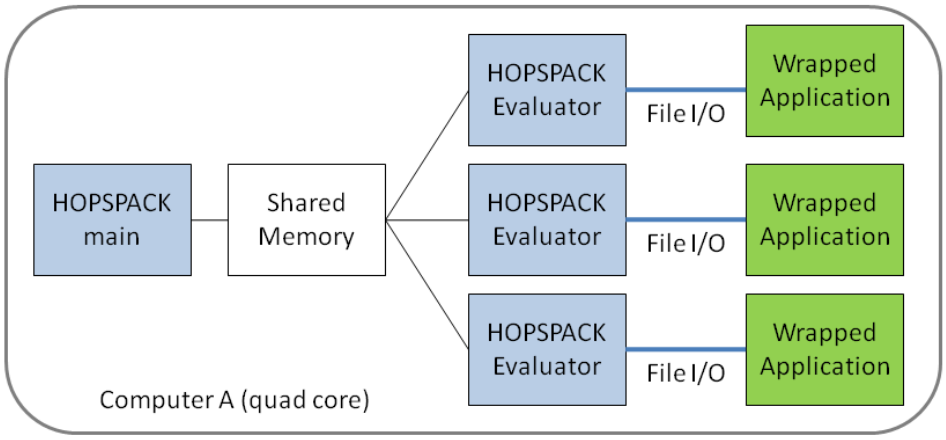


Figure 3. HOPSPACK communication using multithreading.

In this example three copies of the application run in parallel as different processes on the same machine. Four threads run in the HOPSPACK process. Evaluator threads make system calls to the application, and a fourth thread runs the Executor to control evaluations. The Mediator and Citizen solvers also run on the main thread. Execution may be faster if additional cores are available for the application instances.

3.2 Stopping Tests

The Mediator decides when to stop HOPSPACK and what to return as the best point found. The Mediator examines all evaluated points, regardless of which solver submitted it, and keeps the best according to criteria discussed below. HOPSPACK stops if any of the following conditions are met:

- All Citizen solvers are finished.
- The total number of evaluations exceeds parameter `Maximum Evaluations` (p 33). The default value of this parameter imposes no limit on evaluations.
- The best point is feasible and satisfies an objective target set by the user. Parameters `Objective Target` (p 25) and `Objective Percent Error` (p 26) are used to set target values. This is only useful if a practical value for the objective function is known.

A trial point is feasible if it satisfies the bound constraints, linear constraints, and nonlinear constraints defined for the optimization problem. Different tolerances are applied for each type of constraint:

Bound constraints. Variables are feasible if they satisfy the bound constraint exactly.

Linear constraints. Linear equalities and inequalities are satisfied if within a tolerance set by parameter `Active Tolerance` in the "Linear Constraints" sublist (p 29). Computations are made in scaled coordinates (see `Scaling` (p 24)) and normalized using the L_2 norm of the variables and the constraint.

Nonlinear constraints. Nonlinear equalities and inequalities are satifisfied within a tolerance set by parameter `Nonlinear Active Tolerance` in the "Problem Definitions" sublist (p 27). Computations are not scaled.

The Mediator is responsible for choosing the best point that is output when HOPSPACK finishes. This is usually the same "best point" found by GSS and other solvers, but not always. The Mediator first seeks a point that passes the feasibility tests described above. If a feasible point has not been found, then the least infeasible is "best", as measured by the unscaled L_∞ norm of the constraint vector. If a feasible point has been found, then a "better" point must pass the feasibility test and improve on the objective value. Source code is located in HOPSPACK_Mediator.cpp in the method `Mediator::updateBestPoint_()`.

3.3 GSS Overview

Generating Set Search (GSS) in HOPSPACK is an asynchronous implementation of pattern search ideas. An excellent review of pattern search methods and convergence theory is in [5]. GSS with linear constraints is explained in [3] and [6], and GSS with nonlinear constraints in [2]. This overview will provide only a brief outline of the GSS algorithm.

The most basic GSS method addresses problems with continuous variables and only bound constraints. GSS begins with an arbitrary initial point and iterates until stopped. Each iteration

generates a trial point along the positive and negative direction of each coordinate axis. The set of search directions are centered on the current best point (called the "parent" point) and initially extend a certain fixed distance. If one of these trial points improves on the parent, then it becomes the new best point for the next round. If a trial point does worse, then the step size in that direction is reduced to generate a replacement trial point. GSS ends when the step length becomes sufficiently short in every direction emanating from the current best point. Details of this process are explained in the references and in Section 4.7. In particular, parameter `Step Tolerance` (p 36) determines the length of a "sufficiently short" step that stops the algorithm.

The HOPSPACK implementation of GSS is asynchronous in the sense that iterations do not wait for all trial points in a direction set to be evaluated. The algorithm takes action on any partial set of results and is therefore ideal for parallel architectures. Convergence properties are no different for the asynchronous algorithm. An asynchronous implementation is especially tolerant of applications whose run time varies based on the trial point. For example, if certain input regions require much more computation time in the application, GSS will make progress in regions that run fast while waiting for the slower points to evaluate.

GSS can respond to "oracle" inputs provided by another solver. If an evaluation result is better than the current GSS best point, then GSS will adopt this as the new best point and begin searching around it. The feature is controlled by parameter `Ignore Other Points` (p 38).

The addition of linear constraints requires the set of directions to conform with active constraints. GSS honors linear equality constraints at all times, and designates linear inequalities as active if the parent point is within a distance `Epsilon Max` (p 37). Trial points are always feasible with respect to linear constraints. To speed up the search, GSS can guess whether a nearby linear inequality is active and "snap" a trial point to the bound; this feature is controlled by parameters `Snap To Boundary` (p 38) and `Snap Distance` (p 38).

GSS-NLC. The basic GSS solver handles continuous variables with bounds and linear constraints, and was available in APPSPACK 5.0. GSS is extended in HOPSPACK to handle nonlinear constraints in a citizen called GSS-NLC. The algorithm treats violations of nonlinear constraints with a penalty term in the objective [2]. An outer loop of iterations sets the weight of the penalty term and starts a basic GSS citizen to solve a version of the problem without nonlinear constraints. This continues until the subproblem returns a feasible point that also satisfies the `Step Tolerance` (p 36). The inner loop of points generated by the subproblem solver behaves the same as a basic GSS citizen except its initial point and stop criteria are set by the GSS-NLC outer solver.

Performance of GSS-NLC can be heavily influenced by the penalty term and its weight (the weight is also referred to as the penalty parameter). Configuration parameters described in Section 4.8 provide more information. The values of `Final Step Tolerance` (p 42), `Nonlinear Active Tolerance` (p 27), and `Penalty Parameter Increase` (p 41) are particularly important. In general the penalty parameter should be increased rapidly to force feasibility, but on some problems a slightly infeasible path will reach an active nonlinear inequality constraint faster; in this case the penalty parameter should be increased slowly. The user is encouraged to experiment.

The HOPSPACK framework allows a GSS-NLC citizen to create and manage subproblems as separate GSS citizens. This is shown schematically in Figure 1, where a GSS-NLC citizen is connected with a citizen labelled GSS(ρ_k) (the penalty parameter of the subproblem is ρ_k). Subproblems run like any other citizen, but when they finish their result is returned to the parent

citizen that created them. This idea is leveraged to extend GSS for problems with integer variables and multiple start points.

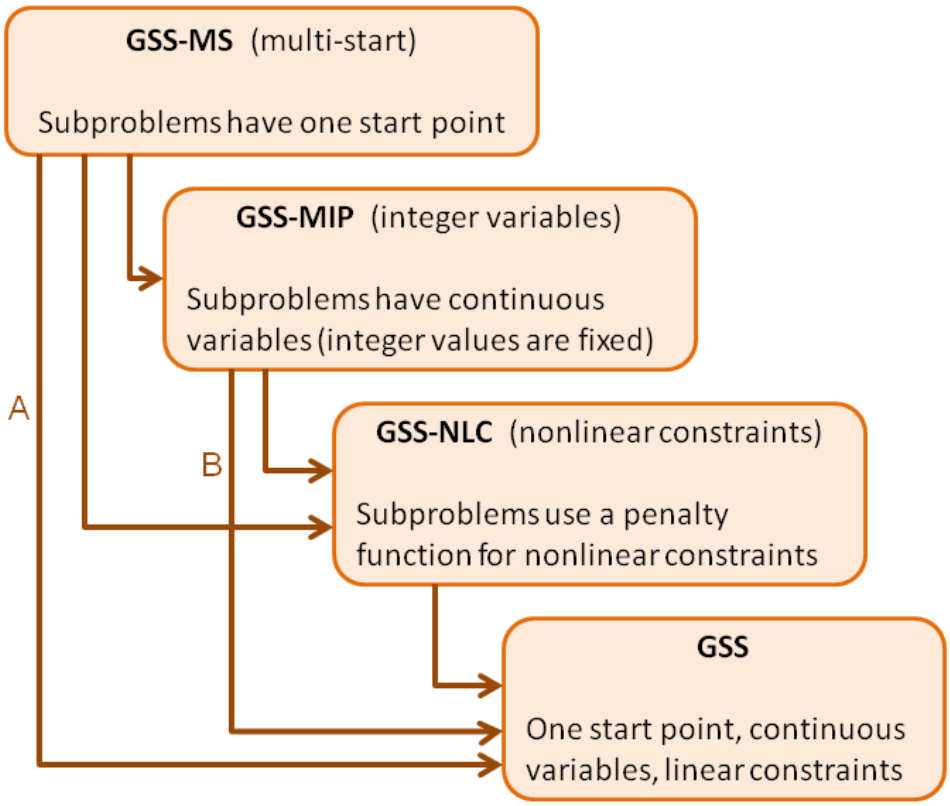


Figure 4. Hierarchy of GSS algorithms, showing how complicated problems are decomposed into simpler subproblems. Arrows "A" and "B" are referenced in the text.

Figure 4 shows how a family of GSS algorithms can address a hierarchy of problem types. Complicated problems at the top follow one or more arrows to reach the basic GSS building block at the bottom. An arrow indicates that a problem is transformed into a sequence of simpler subproblems. For example, path *A* shows that a multi-start problem with continuous variables and linear constraints is solved by creating GSS subproblems, each with a different starting point. The parent GSS-MS collects subproblem results and generates new start points until finished. Path *B* shows that a problem with integer-valued variables and linear constraints is solved by creating GSS subproblems, each treating integer variables as fixed to a specific integer value. The parent GSS-MIP follows a branching tree or other combinatorial strategy to decide how to fix integer variables in each subproblem. In the most complex case, a problem with multiple start points, integer variables, and nonlinear constraints would cascade from GSS-MS to GSS-MIP to GSS-NLC to GSS, creating subproblems of every type.

4 Config Parameters

Execution of HOPSPACK is controlled by a number of configuration parameters. The user provides a text file containing parameters and passes the file name on the command line. HOPSPACK reads the file, parses parameter values, and stores them for use during execution. Most parameters have default values, but an input file is always required to define the optimization problem and how it is evaluated.

As an example, consider solving

$$\begin{array}{lrcl} \min & (x_1-10)^2+(x_2-10)^2+(x_3-10)^2+(x_4-10)^2 & & \\ & & -x_1-x_2-x_3-x_4 & \geq \quad -10 \\ \text{subject to} \quad -1 & \leq \quad x_1-x_2+x_3-x_4 & & \\ & & 2x_1 \quad +2x_3-7x_4 & = \qquad 3 \\ & & 10 \geq x_i \geq -10 & \quad i=1,\ldots,4 \end{array}$$

This problem is provided in the directory examples/2-linear-constraints, where there is more documentation. The input lines below are sufficient to define and solve the problem on HOPSPACK:

```
@ "Problem Definition"
  "Number Unknowns" int 4                     # Number of variables
  "Upper Bounds" vector 4  10  10  10  10     # Variable upper bounds
  "Lower Bounds" vector 4 -10 -10 -10 -10     # Variable lower bounds
  "Initial X"    vector 4  -1   1  -1  -1     # Initial point
@@
@ "Linear Constraints"
  "Inequality Matrix" matrix 2 4              # 2 ineq, 4 variables
   -1 -1 -1 -1
    1 -1  1 -1
  "Inequality Lower" vector 2  -10 DNE        # Lower bounds on 2 ineqs
  "Inequality Upper" vector 2  DNE  -1        # Upper bounds on 2 ineqs
  "Equality Matrix" matrix 1 4                # 1 equality, 4 variables
   2.0  0  2.0  -0.7e+1
  "Equality Bounds" vector 1 3                # Right-hand side of eqs
@@
@ "Evaluator"                                 # Program computing the obj
  "Executable Name" string "linear_constraints"
@@
@ "Mediator"
  "Citizen Count" int 1                       # One citizen to start
@@
@ "Citizen 1"                                 # Citizen name
  "Type" string "GSS"                         # Generalized Set Search
@@
```

As the example illustrates, parameters are organized into sublists that begin with the `@` character and a keyword name; for example, `@ "Problem Definition"`. Sublists end with a line containing `@@`. Sublists can appear in any order.

Within a sublist are one or more parameters, each on a separate line. Parameters can appear in any order within a sublist. Parameter names are scoped within a sublist, so if the same parameter name appears in two sublists, it is actually two separate parameters with separate values. Each line begins with the parameter name (a case-sensitive string), the type of the parameter (`int`, `double`, etc.), and then the value. The file can contain empty lines, leading whitespace, and comments beginning with the `#` character.

- String, integer, and double precision parameters are specified as follows:

```
"String Parameter Name" string "string value"
"Integer Parameter Name" int integer_value
"Double Parameter Name" double double_value
```

  The double precision value can be in floating point or scientific format. HOPSPACK also accepts the string "`DNE`" for a double precision value, which indicates the value "does not exist". For instance, in the example above `DNE` is used to show the first inequality constraint has no upper bound.

- Boolean values are specified as follows:

```
"Boolean Parameter Name" bool value
```

  The value is the word "true" or "false", without quotes, or one of the equivalent words "TRUE", "True", "T" (similarly, for "false").

- Vector parameters are specified with a length (N) and then double precision values, all on one line:

```
"Vector Parameter Name" vector N double_1 ... double_N
```

- Matrix parameters are specified with a number of rows (M) and columns (N). Double precision values for each row are given on consecutive lines:

```
"Matrix Parameter Name" matrix M N
  double_1,1 ... double_1,N
  double_2,1 ... double_2,N
            ...
  double_M,1 ... double_M,N
```

- A special character vector parameter is used to specify variable types. It is a vector of single-letter characters:

```
"Character Vector Parameter Name" charvec N char_1 ... char_N
```

The next section (4.1) describes the assumptions made when formulating an optimization problem. A quick reference guide to all the parameters is provided in Section 4.2. The remaining sections describe each parameter in HOPSPACK, grouped by sublist.

4.1 Defining the Optimization Problem

Certain conventions are assumed when defining an optimization problem for HOPSPACK. The main points are described below. Documentation of the configuration parameters should also be consulted, especially the "Problem Definition" sublist (Section 4.3) and the "Linear Constraints" sublist (Section 4.4).

- Variables appear in an arbitrary order determined by the user, but that order must be maintained when listing upper and lower bounds, variable types, scaling factors, and any initial start point.
- If the objective cannot be evaluated at a point, then the application should return the string "`DNE`" to indicate the value does not exist. Also return `DNE` for any nonlinear constraints that cannot be evaluated at a point.
- Linear equalities and inequalities are separated using different parameters (see Section 4.4). Constraints within each set appear in an arbitrary order determined by the user, but that order must be maintained when listing the matrix coefficients and bounds. A single inequality can have a lower and upper bound, or just one bound.
- Nonlinear equalities and inequalities are separated. Constraints within each set appear in an arbitrary order determined by the user, but that order must be maintained when returning evaluation values or defining values at an initial start point. Nonlinear equalities should be defined to equal zero; i.e., a value of zero at x indicates the equality is satisfied:

  $$c_{\mathcal{E}}(x) = 0.$$

  Nonlinear inequalities should be defined as greater than zero:

  $$c_{\mathcal{I}}(x) \geq 0.$$

  Hence, a negative value at x indicates that an inequality is violated, and a nonnegative value indicates feasibility. A nonlinear inequality with an upper and lower bound should be written as two inequalities; for example,

  $$8 \;\geq\; x^2 + y^2 \;\geq\; 5 \quad \rightarrow \quad \left\{ \begin{array}{rcl} 8 - x^2 - y^2 & \geq & 0 \\ x^2 + y^2 - 5 & \geq & 0 \end{array} \right.$$

4.2 Quick Reference for Config Parameters

The tables below list all configuration parameters in alphabetical order, grouped by sublist.

Problem Definition sublist (Section 4.3)

Parameter Name	*Type*	*Default Value*
Display (p 28)	`int`	`0` (no display of problem definition)
Initial F (p 27)	`vector`	`DNE` (no initial values)
Initial Nonlinear Eqs (p 27)	`vector`	`DNE` (no initial values)
Initial Nonlinear Ineqs (p 27)	`vector`	`DNE` (no initial values)
Initial X (p 27)	`vector`	no initial point
Lower Bounds (p 24)	`vector`	no lower bounds (all equal `DNE`)
Nonlinear Active Tolerance (p 27)	`double`	`1.0e-7`
Number Nonlinear Eqs (p 26)	`int`	`0`
Number Nonlinear Ineqs (p 26)	`int`	`0`
Number Objectives (p 25)	`int`	`1`
Number Unknowns (p 24)	`int`	none (parameter is required)
Objective Percent Error (p 26)	`double`	`DNE` (no stop test based on percent error)
Objective Target (p 25)	`double`	`DNE` (no target)
Objective Type (p 25)	`string`	`"Minimize"`
Scaling (p 24)	`vector`	automatic scaling if possible
Upper Bounds (p 24)	`vector`	no upper bounds (all equal `DNE`)
Variable Types (p 25)	`charvec`	all variables continuous

Linear Constraints sublist (Section 4.4)

Parameter Name	*Type*	*Default Value*
Active Tolerance (p 29)	`double`	`1.0e-12`
Display (p 30)	`int`	`0` (no display of constraints)
Equality Bounds (p 28)	`vector`	no equality constraints
Equality Matrix (p 28)	`matrix`	no equality constraints
Inequality Lower (p 28)	`vector`	no inequality constraints
Inequality Matrix (p 29)	`matrix`	no inequality constraints
Inequality Upper (p 29)	`vector`	no inequality constraints

Evaluator sublist (Section 4.5)

Parameter Name	*Type*	*Default Value*
Debug Eval Worker (p 31)	`bool`	`false`
Evaluator Type (p 30)	`string`	`"System Call"`
Executable Name (p 30)	`string`	`"a.out"`
File Precision (p 31)	`int`	`14`
Input Prefix (p 30)	`string`	`"input"`
Output Prefix (p 31)	`string`	`"output"`
Save IO Files (p 31)	`bool`	`false`

Mediator sublist (Section 4.6)

Parameter Name	*Type*	*Default Value*

Cache Comparison Tolerance (p 33)	`double`	2 * machine epsilon ($\approx 4.4 \times 10^{-16}$)
Cache Enabled (p 33)	`bool`	`true`
Cache Input File (p 33)	`string`	no input file
Cache Output File (p 34)	`string`	no output file
Cache Output Precision (p 34)	`int`	`14`
Citizen Count (p 32)	`int`	none (parameter is required)
Display (p 35)	`int`	`2`
Maximum Evaluations (p 33)	`int`	`-1` (unlimited)
Maximum Exchange Return (p 34)	`int`	`1000`
Minimum Exchange Return (p 34)	`int`	`1`
Number Processors (p 32)	`int`	none (parameter required with MPI)
Number Threads (p 32)	`int`	none (parameter required if multithreaded)
Precision (p 35)	`int`	`3`
Reserved Citizen Workers (p 32)	`int`	`0`
Solution File (p 35)	`string`	no solution file
Solution File Precision (p 35)	`int`	`14`
Synchronous Evaluations (p 34)	`bool`	`false`

Citizen GSS sublist (Section 4.7)

Parameter Name	*Type*	*Default Value*
Add Projected Compass (p 38)	`bool`	`false`
Add Projected Normals (p 39)	`bool`	`true`
Citizen Priority (p 36)	`int`	`1` (highest priority)
Contraction Factor (p 36)	`double`	`0.5`
Display (p 40)	`int`	`0` (no display of GSS operation)
Epsilon Max (p 37)	`double`	`2 * Step Tolerance`
Ignore Other Points (p 38)	`bool`	`false`
Initial Step (p 37)	`double`	`1.0`
Maximum Evaluations (p 39)	`int`	`-1` (unlimited)
Maximum Queue Size (p 39)	`int`	`0`
Minimum Step (p 37)	`double`	`2 * Step Tolerance`
Penalty Function (p 39)	`string`	`"L2 Squared"`
Penalty Parameter (p 39)	`double`	`1.0`
Penalty Smoothing Value (p 39)	`double`	`0.0`
Snap Distance (p 38)	`double`	`Step Tolerance / 2`
Snap To Boundary (p 38)	`bool`	`false`
Step Tolerance (p 36)	`double`	`0.01`
Sufficient Improvement Factor (p 37)	`double`	`0.01`
Type (p 36)	`string`	`"GSS"`
Use Random Order (p 38)	`bool`	`true`

Citizen GSS-NLC sublist (Section 4.8)

Parameter Name	*Type*	*Default Value*
Display (p 43)	`int`	`0` (no display of GSS-NLC operation)
Display Subproblem (p 44)	`int`	`0` (no display of subproblem operation)
Final Step Tolerance (p 42)	`double`	`0.001`
Ignore Other Points (p 43)	`bool`	`false`

Initial Step Tolerance (p 42)	`double`	`0.1`
Maximum Evaluations (p 43)	`int`	`-1` (unlimited)
Max Subproblem Evaluations (p 43)	`int`	`-1` (unlimited)
Penalty Function (p 40)	`string`	`"L2 Squared"`
Penalty Parameter (p 41)	`double`	`1.0`
Penalty Parameter Increase (p 41)	`double`	`2.0`
Penalty Parameter Maximum (p 41)	`double`	`1.0e+8`
Penalty Smoothing Value (p 42)	`double`	`0.0`
Smoothing Value Decrease (p 42)	`double`	`0.5`
Smoothing Value Minimum (p 42)	`double`	`1.0e-5`
Step Tolerance Decrease (p 42)	`double`	`0.5`
Type (p 40)	`string`	`"GSS-NLC"`

4.3 Problem Definition Sublist Parameters

This sublist defines the optimization objective, the variables, bounds on the variables, scaling factors for each variable, a starting point for algorithms to use, and the number and type of nonlinear constraints. The sublist has one required parameter: `Number Unknowns`. All other parameters have default values that assume the problem is an unconstrained minimization of continuous variables.

Number Unknowns. Specifies the number of optimization variables in the problem, including continuous and integer-valued unknowns. The parameter value must be a positive integer.

Type: `int`

Default: none (the parameter is required)

Example: `"Number Unknowns" int 4`

Lower Bounds. Specifies a lower bound for each variable. If a variable has no lower bound, then use the value `DNE` in that position of the vector. It is possible, but not recommended, to define a simple variable bound as an inequality in the "Linear Constraints" sublist. The number of values supplied by the parameter must equal the number of unknowns.

Type: `vector`

Default: no lower bounds (all values equal `DNE`)

Upper Bounds. Specifies an upper bound for each variable. If a variable has no upper bound, then use the value `DNE` in that position of the vector. It is possible, but not recommended, to define a simple variable bound as an inequality in the "Linear Constraints" sublist. The number of values supplied by the parameter must equal the number of unknowns.

Type: `vector`

Default: no upper bounds (all values equal `DNE`)

Scaling. Variables with widely different ranges can substantially slow the convergence of GSS and other algorithms. Scaling allows algorithms to correct for the discrepancy, and is considered essential for good performance.

The parameter states the expected range of each variable, in either absolute or relative terms. If no parameter is specified, then a default value of one is used (i.e., all variables are scaled equally). However, HOPSPACK insists that the problem define either scaling or upper and lower bounds on each variable. This ensures citizens have some information about the range of variables for their algorithm; for example, GSS chooses a default value for `Initial Step` (p 37) based on the scaling or variable bounds.

The GSS algorithm makes many internal calculations in scaled coordinates, and its behavior can change based on the scaling. For example, the `Step Tolerance` (p 36) determines when a local

solution is found. If scaling is increased for a problem, then the step tolerance may need to decrease in order to reach a solution with the same accuracy.

If the user has no information about variable scaling but is confident of the upper and lower bounds, then often a reasonable guess for scaling is the range of the variable:

$$\text{scale}_i = u_i - l_i$$

The number of values supplied by the parameter must equal the number of unknowns.

Type: `vector`

Default: automatic scaling if possible, else the parameter is required

Variable Types. Specifies the type of each variable with a vector of character values. Possible character codes are `C` (or `c`) for real-valued continuous variables, `I` (or `i`) for integer-valued variables that can be relaxed to continuous values, and `O` (or `o`) for ordinal variables that can only take integer values. Variables of type `I` and `O` are both integral at the solution, but an application with type `I` variables will accept continuous values; for example, mixed integer programming problems that can be relaxed to a continuous linear programming problem have variables of type `I`. Type `O`, known as ordinal or categorical variables, require integer values at all evaluations points. The number of values supplied by the parameter must equal the number of unknowns.

Type: `charvec`

Default: all variables continuous

Example: `"Variable Types" charvec 3 C C I`

Number Objectives. Specifies the number of optimization objective functions. A value of one indicates there is a single objective function. A value of zero means there is no objective and the goal is to find a feasible point. A value greater than one indicates there are multiple objectives.

Type: `int`

Default: `1`

Objective Type. Specifies the goal of the optimization, which must be one of the keywords `"Minimize"` or `"Maximize"`. The parameter is ignored if `Number Objectives` is zero.

Type: `string`

Default: `"Minimize"`

Objective Target. Specifies a satisfactory value for the objective function to reach. If HOPSPACK finds a feasible point with an objective equal to or better than the target value, then execution stops immediately. If the `Objective Type` parameter is `"Minimize"`, then HOPSPACK stops when the objective value is less than or equal to the target; if `"Maximize"`, then it stops

when the objective is greater than or equal to the target. (Note: in APPSPACK this parameter was called "Known Global Minimum".)

In many problems the optimal value of the objective is not known. In this case, leave the parameter undefined. HOPSPACK can still find a solution, but will probably take more iterations compared to a known target, because it will make extra evaluations to confirm that neighboring points are no better.

Use the parameter `Objective Percent Error` to instruct HOPSPACK to stop when within a certain range of the target value.

Type: `double`

Default: `DNE` (no target)

Objective Percent Error. Specifies a desired value for the objective function to reach, in terms of parameter `Objective Target`. Suppose the current objective value is f and the target value is f_T. If f reaches or exceeds the target, then HOPSPACK stops, as explained in the description of parameter `Objective Target`. If f does not reach the target but is within a percentage of it as defined in the equation below, then HOPSPACK stops. Pseudocode for the combined stop test follows:

If f reaches or exceeds `Objective Target`	Then stop
If $100 \times \dfrac{\lvert f - f_T \rvert}{\max\{10^{-4}, \lvert f_T \rvert\}} \leq$ `Objective Percent Error`	Then stop

The parameter value cannot be negative.

Type: `double`

Default: `DNE` (no stop test based on percent error)

Example: `"Objective Percent Error" double 2.5` (stop if within 2.5%)

Number Nonlinear Eqs. Specifies the number of nonlinear equality constraints in the problem. Nonlinear equalities are defined in the form $c(x) = 0$, so a value of zero indicates feasibility.

Type: `int`

Default: `0`

Number Nonlinear Ineqs. Specifies the number of nonlinear inequality constraints in the problem. Nonlinear inequalities are defined in the form $c(x) \geq 0$, so a negative value indicates the constraint is violated.

Type: `int`

Default: `0`

Nonlinear Active Tolerance. Specifies the maximum violation beyond which a nonlinear equality or inequality constraint is considered active. For example, if an equality constraint evaluates to a value larger than the tolerance, then the point is interpreted to be infeasible. Note that the nonlinear tolerance is defined and treated differently than the `Active Tolerance` parameter in the "Linear Constraints" sublist (p 29).

Type: `double`

Default: `1.0e-7` (10^{-7})

Initial X. Initial point for optimization to begin at. If any variable violates its upper or lower bound, then it is moved to the bound to become feasible. If the initial point violates linear constraints, then a least squares subproblem is automatically solved to project it onto the constraints, providing a feasible initial point. If an initial point is not defined, then each citizen is at liberty to define a default initial point. The number of values supplied by the parameter must equal the number of unknowns.

Type: `vector`

Default: no initial point

Initial F. Provides known objective values at the initial point specified by parameter `Initial X`. If no initial point is given, or if the point is moved by HOPSPACK to satisfy constraints, then this parameter is ignored. If parameter `Initial X` is given and this parameter is not, then HOPSPACK will evaluate the initial point before any other trial points.

Type: `vector` (initial objective values)

Default: values not known

Initial Nonlinear Eqs. Provides known values for all nonlinear equality constraints at the initial point specified by parameter `Initial X`. Nonlinear equalities are defined in the form $c(x) = 0$, so a value of zero indicates feasibility. If no initial point is given, or if the point is moved by HOPSPACK to satisfy constraints, then this parameter is ignored. If the problem has nonlinear equalities, parameter `Initial X` is given and this parameter is not, then HOPSPACK will evaluate the initial point before any other trial points. The number of values supplied by the parameter must equal the value of `Number Nonlinear Eqs`.

Type: `vector` (initial equality constraint values)

Default: values not known

Initial Nonlinear Ineqs. Provides known values for all nonlinear inequality constraints at the initial point specified by parameter `Initial X`. Nonlinear inequalities are defined in the form $c(x) \geq 0$, so a negative value indicates the constraint is violated. If no initial point is given, or if the point is moved by HOPSPACK to satisfy constraints, then this parameter is ignored. If the problem has nonlinear inequalities, parameter `Initial X` is given and this parameter is not, then

HOPSPACK will evaluate the initial point before any other trial points. The number of values supplied by the parameter must equal the value of `Number Nonlinear Ineqs`.

Type: `vector` (initial inequality constraint values)

Default: values not known

Display. Specifies whether to print the problem definition to the console. Printing of the problem definition can happen at the start of HOPSPACK execution, and when the GSS citizen initializes. Possible values are:

- `0` display nothing
- `1` display problem summary
- `2` display all details, including bounds and type of each variable

Type: `int`

Default: `0` (display nothing)

4.4 Linear Constraints Sublist Parameters

This sublist defines linear equality and inequality constraints in the problem, and a tolerance for determining when a constraint is active. The sublist can be omitted if the problem has no linear constraints. Simple bounds on a single variable should be defined in the "Problem Definition" sublist (Section 4.3).

Equality Bounds. Provides the right-hand side for all linear equality constraints. The order of values matches the order of the rows in `Equality Matrix`. See the example at the beginning of Section 4.

Type: `vector`

Default: no equality constraints

Equality Matrix. Provides the left-hand side coefficients for all linear equality constraints. The order of matrix rows matches the order of the values in `Equality Bounds`. The number of matrix columns must equal the number of unknowns. Coefficients are listed in dense notation; that is, a zero value must be given for any missing variable terms. See the example at the beginning of Section 4.

Type: `matrix`

Default: no equality constraints

Inequality Lower. Provides the lower bound for all linear inequality constraints. The order of values matches the order of the rows in `Inequality Matrix`. If an inequality has no lower bound, then use the value `DNE` in that position of the vector. See the example at the beginning of Section 4.

Type: `vector`

Default: no inequality constraints

Inequality Upper. Provides the upper bound for all linear inequality constraints. The order of values matches the order of the rows in `Inequality Matrix`. If an inequality has no upper bound, then use the value `DNE` in that position of the vector. See the example at the beginning of Section 4.

Type: `vector`

Default: no inequality constraints

Inequality Matrix. Provides the coefficients for all linear inequality constraints. The order of matrix rows matches the order of the values in `Inequality Lower` and `Inequality Upper`. The number of matrix columns must equal the number of unknowns. Coefficients are listed in dense notation; that is, a zero value must be given for any missing variable terms. See the example at the beginning of Section 4.

Type: `matrix`

Default: no inequality constraints

Active Tolerance. Specifies the distance over which an equality or inequality boundary is considered active. For example, a point located beyond the active tolerance of a linear equality constraint is interpreted to be infeasible. Several other HOPSPACK operations depend on the tolerance value: the computation of a step length to the bounds (see `HOPSPACK_LinConstr::maxStep()`), the choice of a set of active constraints to "snap" onto (see `HOPSPACK_LinConstr::formSnapSystem()`), the threshold for declaring a constraint to be degenerate (see `HOPSPACK_Matrix::nullspace()`), the construction of normal and tangent cones for generating search directions in the GSS citizen (see methods in `HOPSPACK_GssDirections`), and the solution of a least squares subproblem to project infeasible trial points onto the linear constraints (see methods in `HOPSPACK_SolveLinConstrProj`).

In most cases the active tolerance is applied against a scaled distance to a constraint. Therefore, if `Scaling` (p 24) is large for some variables, the active tolerance may need to increase beyond its default value.

Note that the linear tolerance is defined and treated differently than the `Nonlinear Active Tolerance` parameter in the "Problem Definition" sublist (p 27).

The problem in examples/2-linear-constraints illustrates the importance of this parameter. If the tolerance is made smaller than the default, then some trial points computed by projection onto the constraints are judged to be infeasible. As an example, try changing the tolerance to `4.0e-16`

(exact results of this experiment depend on the machine architecture).

Type: `double`

Default: `1.0e-12` (10^{-12})

Display. Specifies whether to print the linear constraints to the console. Printing of the problem definition can happen at the start of HOPSPACK execution, and when the GSS citizen initializes. Possible values are:

- `0` display nothing
- `1` display linear constraints summary
- `2` display all details, including the constraint matrices

Type: `int`

Default: `0` (display nothing)

4.5 Evaluator Sublist Parameters

This sublist determines how functions are evaluated. Every optimization problem must provide a way for HOPSPACK to compute its objective and nonlinear constraints. Read Section 5 for an explanation of how HOPSPACK invokes an evaluation.

Evaluator Type. Specifies the evaluator type with a special string value. Possible values are `"System Call"` and `"AMPL Call"`.

Type: `string`

Default: `"System Call"`

Executable Name. Specifies the name of an executable program that HOPSPACK calls to evaluate optimization problem data at a trial point. The program is passed certain command line arguments and uses files to communicate inputs and outputs. See Section 5 for information on writing the executable.

This parameter is used only if the `Evaluator Type` parameter is `"System Call"`. Note the executable name should be a full path if the user's `PATH` environment variable does not include the location of the executable.

Type: `string`

Default: `"a.out"`

Input Prefix. Specifies the prefix of file names that are used to pass input data to the executable program defined by parameter `Executable Name`. The full file name has the form

input_prefix.NNN_TT, where NNN is a unique integer tag assigned to the trial point and TT is the type of information requested.

This parameter is used only if the `Evaluator Type` parameter is `"System Call"`.

Type: `string`

Default: `"input"`

File Precision. Specifies the number of digits after the decimal point for each floating point number written to the file named with parameter `Input Prefix`.

This parameter is used only if the `Evaluator Type` parameter is `"System Call"`.

Type: `int`

Default: `14`

Output Prefix. Specifies the prefix of file names that are used to return output from the executable program defined by parameter `Executable Name`. The full file name has the form output_prefix.NNN_TT, where NNN is a unique integer tag assigned to the trial point and TT is the type of information requested.

This parameter is used only if the `Evaluator Type` parameter is `"System Call"`.

Type: `string`

Default: `"output"`

Save IO Files. If true, then the data files formed with parameters `Input Prefix` and `Output Prefix` are saved. If false, then the files are deleted when no longer needed. Keep in mind that two files will be created for every trial point that HOPSPACK evaluates.

This parameter is used only if the `Evaluator Type` parameter is `"System Call"`.

Type: `bool`

Default: `false`

Debug Eval Worker. If true, then debugging messages are written during execution of the HOPSPACK evaluator. On the MPI version of HOPSPACK, evaluations usually take place on distributed machines, and this is where the debugging messages will appear.

Type: `bool`

Default: `false`

4.6 Mediator Sublist Parameters

This sublist controls operation of the HOPSPACK framework, including the number of citizens and citizen workers, the Cache of evaluated trial points, the manner in which points are exchanged with the Executor, and the maximum number of evaluations. The sublist has one required parameter: `Citizen Count`.

Citizen Count. Specifies the number of citizens defined in the configuration parameters file. The value must match the number of citizen sublists. Some citizens may dynamically create child citizens during HOPSPACK execution, but these dynamic citizens are not part of the `Citizen Count`.

Type: `int`

Default: none (the parameter is required)

Number Processors. Specifies the number of processors to be used with the MPI version of HOPSPACK. One processor will be dedicated to the main loop that runs the Mediator and Citizens. Additional processors will be allocated to match the value of `Reserved Citizen Workers`. The remaining processors are used to evaluate trial points. There must be enough processors to support at least one evaluation worker. The value should not exceed the number of processors passed to MPI when HOPSPACK is invoked.

The parameter is ignored if the HOPSPACK executable does not use MPI.

Type: `int`

Default: none (the parameter is required with an MPI executable)

Number Threads. Specifies the number of threads to be started with the multithreaded version of HOPSPACK. One thread will be dedicated to the main loop that runs the Mediator and Citizens. Additional threads will be allocated to match the value of `Reserved Citizen Workers`. The remaining threads are used to evaluate trial points. There must be enough threads to support at least one evaluation worker.

If the number of threads exceeds the number of CPU cores, then some evaluation workers will do much less work than others, and overall efficiency usually decreases.

The parameter is ignored if the HOPSPACK executable does not use multithreading.

Type: `int`

Default: none (the parameter is required with a multithreaded executable)

Reserved Citizen Workers. (This parameter is reserved for future use.)

Type: `int`

Default: `0`

Maximum Evaluations. Specifies how many trial points should be evaluated before halting. The limit applies to executed function evaluations and does not include trial points found in the Cache. A negative value means there is no limit on the number of evaluations.

Type: `int`

Default: `-1` (unlimited)

Cache Enabled. If true, then evaluated points are saved in the Cache and used to prevent duplicate evaluation requests. If false, then the Cache is not even constructed.

Type: `bool`

Default: `true`

Cache Comparison Tolerance. Specifies how far apart two points must be for the Cache to consider them distinct. The tolerance is measured as a scaled distance, and applies to each component of a point. If the two points are vectors a and b, the scaling vector is s, and the comparison tolerance is denoted by τ, then the Cache considers a and b to be distinct if

$$|a_i - b_i| > \tau s_i \text{ for some } i = 1, \ldots, n$$

or, equivalently,

$$\left\| \frac{a_i - b_i}{s_i} \right\|_\infty > \tau.$$

Type: `double`

Default: 2 * machine epsilon ($\approx 4.4 \times 10^{-16}$)

Cache Input File. Specifies the name of an input file containing evaluated points for the current problem definition. The file is read when HOPSPACK starts and all points are loaded into the Cache, using the current `Cache Comparison Tolerance` to determine when points are distinct. The file format is text, and is identical to a file produced by setting `Cache Output File` and running HOPSPACK. Users must be careful that the input file corresponds to the current problem definition.

Note that the HOPSPACK file format is not compatible with the older APPSPACK cache file format.

Type: `string`

Default: no input file

Cache Output File. Specifies the name of an output file to write all evaluated points. Points and their evaluations are appended to the file; hence, users must be careful that any existing data corresponds to the same problem definition. Only distinct points are written, based on the current value of `Cache Comparison Tolerance`. The file format is text, and is identical to the format expected for `Cache Input File`. The number of digits written for each floating point number is determined by `Cache Output File Precision`.

Note that the HOPSPACK file format is not compatible with the older APPSPACK cache file format.

Type: `string`

Default: no output file

Cache Output File Precision. Specifies the number of digits after the decimal point for each floating point number written to `Cache Output File`.

Type: `int`

Default: `14`

Synchronous Evaluations. Determines the primary behavior of a Conveyor iteration. If true, then all trial points submitted by citizens are evaluated in a single Conveyor iteration. Thus, from the point of view of a citizen, the list of trial points submitted is returned on the next call with evaluation information, as though the citizen plus evaluations were happening in a single execution path. If the parameter is false, then the number of points submitted and returned by a Conveyor iteration is governed by the parameters `Maximum Exchange Return` and `Minimum Exchange Return`. From the point of view of a citizen, submitted trial points are returned asynchronously at an unknown future iteration.

Type: `bool`

Default: `false`

Maximum Exchange Return. Specifies the maximum number of evaluated points returned during a single iteration of the Conveyor. The actual number returned may be less, depending on asynchronous behavior of the architecture. The value must be positive and no smaller than the value of `Minimum Exchange Return`. The parameter is ignored if `Synchronous Evaluations` is set `true`.

Type: `int`

Default: `1000`

Minimum Exchange Return. Specifies the minimum number of trial points submitted for evaluation during a single iteration of the Conveyor. The actual number submitted may be less if

citizens have not queued the minimum number at the start of a Conveyor iteraton. The value must be positive. The parameter is ignored if `Synchronous Evaluations` is set `true`.

Type: `int`

Default: `1`

Solution File. Provides the name of an output file to write the final solution. The point and its evaluations are appended to the file; hence, users must be careful that any existing data corresponds to the same problem definition. The number of digits written for each floating point number is determined by `Solution File Precision`.

Type: `string`

Default: no solution file

Solution File Precision. Specifies the number of digits after the decimal point for each floating point number written to `Solution File`.

Type: `int`

Default: `14`

Precision. Specifies the number of digits after the decimal point when printing numbers associated with vectors and matrices. For example, the parameter controls the format of evaluated points printed during execution. The parameter may also control the output from some citizens.

Type: `int`

Default: `3`

Display. Specifies how much information to print about operation of the HOPSPACK framework. Citizens must provide their own `Display` parameter to print their internal operations. Possible values are:

- `1` display the final solution
- `2` display the final solution and input parameters
- `3` display the above, and all evaluated points
- `4` display the above, and all trial points
- `5` display the above, and execution details

Type: `int`

Default: `2` (display final solution and input parameters)

4.7 Citizen GSS Sublist Parameters

This sublist determines operational specifics of a Generating Search Set algorithm. The citizen is designed for problems with continuous variables, a single start point, and at most linear constraints (no nonlinear constraints). Like all citizens, the `Type` parameter is required to identify the citizen. See Section 3.3 for more information about the GSS algorithm.

Type. Specifies the citizen type with a special string value.

Type: `string`

Must be: `"GSS"`

Citizen Priority. Specifies the priority of submitted trial points. All citizens are assigned a priority, and the Conveyor submits points from higher priority citizens before lower priority citizens. The value must be between 1 and 10, with 1 being the highest priority.

Type: `int`

Default: `1`

Step Tolerance. Determines the smallest steps that GSS will try before stopping. When a trial point in a particular search direction fails to improve upon its parent point (see `Sufficient Improvement Factor`), then the step length is decreased by the `Contraction Factor`. GSS continues to try shorter steps until they become smaller than the value of `Step Tolerance`. When all search directions from the current "best point" have contracted to a length smaller than `Step Tolerance`, then GSS identifies the point as a solution and stops.

Step length is measured as a scaled distance, based on the `Scaling` parameter in the "Problem Definition" sublist (p 24). If s_i is the scaling factor for variable x_i, then GSS stops searching along x_i when

$$|(\text{step length})_i| < s_i(\texttt{Step Tolerance}).$$

If scaling is unity in every direction, then `Step Tolerance` can be thought of as percent error over a unit cube centered on the point identified as a solution.

Note that `Step Tolerance` should not be smaller than the `Cache Comparison Tolerance` parameter in the "Mediator" sublist (p 33); otherwise, small steps may be assigned the value of a previously evaluated neighbor.

Type: `double`

Default: `0.01`

Contraction Factor. Specifies the reduction in step length in a particular search direction after a trail point is rejected. A point is rejected if the objective does not improve enough com-

pared with its parent (see `Sufficient Improvement Factor` for more details). The value must be positive and less than one.

Type: `double`

Default: `0.5`

Sufficient Improvement Factor. Determines the amount of improvement that the objective of a trial point must make in comparison with its parent point in order to be accepted. The necessary improvement ρ is computed as

$$\rho = \alpha(\text{step length})^2$$

where α is the `Sufficent Improvement Factor`. Convergence of the GSS method is guaranteed (with certain assumptions about the objective function) when $\alpha > 0$. The value can also be set to zero.

Type: `double`

Default: `0.01`

Initial Step. Specifies the initial, scaled step length when generating a new trial point. Ideally, the initial step covers the expected distance from start point to a solution. If unsure, it is best to make the value too large rather than too small.

If no value is supplied and the `Scaling` parameter in the "Problem Definition" sublist (p 24) is defined, then the default value of `Initial Step` is 1. If no value is supplied, `Scaling` is not provided, and the default assignment of `Scaling` values equals one, then GSS computes a value equal to half the maximum distance between upper and lower variable bounds.

Type: `double`

Default: `1.0`

Minimum Step. Specifies the smallest size step that can be taken when generating a new GSS trial point. The value must be greater than `Step Tolerance`.

Type: `double`

Default: `2 * Step Tolerance`

Epsilon Max. Defines the maximum allowed radius about a point when determining whether a linear constraint is active. The active set of constraints influences the generation of search directions and new trial points.

Type: `double`

Default: `2 * Step Tolerance`

Snap To Boundary. If true, then every newly generated point is "snapped" onto nearby linear constraints. A constraint is considered "nearby" if its scaled range is less than or equal to `Snap Distance`. Snapping can speed the discovery of a solution point at a corner of the feasible region, and provides more accuracy when there is such a solution. Snapping requires that HOPSPACK be configured with LAPACK (see Section 6.3).

Type: `bool`

Default: `false`

Snap Distance. Specifies the search radius for finding nearby linear constraints that a point may "snap" onto. The radius is interpreted as a scaled distance. The parameter is ignored if `Snap To Boundary` is set `false`. (Note: in APPSPACK this parameter was called "Bounds Tolerance".)

Type: `double`

Default: `Step Tolerance / 2`

Use Random Order. If true, then trial points are submitted to the Conveyor in random order. When evaluations are made asynchronously, random ordering reduces the tendency for initial search directions to be overemphasized.

Type: `bool`

Default: `true`

Ignore Other Points. Specifies whether the GSS citizen ignores evaluated points that were generated by other citizens. If false, then GSS will consider all evaluated points and center the search around the best point found, regardless of how that point was generated. If true, then the citizen determines a best point only from the set of candidates that it generated.

Type: `bool`

Default: `false`

Add Projected Compass. Specifies whether to include certain directions when generating a pattern search. If true, then standard compass directions are added after being projected onto the null space of the active linear constraints. Compass directions are vectors aligned with each coordinate axis, both positive and negative:

$$\{\pm e_1, \pm e_2, \ldots, \pm e_n\}$$

Type: `bool`

Default: `false`

Add Projected Normals. Specifies whether to include certain directions when generating a pattern search. If true, then generators for the normal cone are added after being projected onto the null space of any linear equality constraints.

Type: `bool`

Default: `true`

Maximum Queue Size. Specifies the maximum number of points that remain "pending" in a GSS citizen queue. The Conveyor asks a citizen for new trial points at the beginning of each Conveyor iteration. If the number of previously submitted points still waiting for evaluation exceeds `Maximum Queue Size`, then the GSS citizen will discard excess points, and then add any newly generated trial points. Note that the number of points submitted by a GSS citizen can exceed the parameter value.

Type: `int`

Default: `0`

Maximum Evaluations. Specifies how many trial points should be evaluated before the GSS citizen halts. The limit applies to executed function evaluations and does not include trial points found in the Cache. The parameter limits evaluations by each instance of a GSS citizen, whether created directly from the configuration file or called as a subproblem on behalf of another citizen. A negative value means there is no limit on the number of evaluations.

Type: `int`

Default: `-1` (unlimited)

Penalty Function. The parameter is typically used when the citizen is solving a subproblem on behalf of another citizen. For more information, see the "Penalty Function" parameter in the "Citizen GSS-NLC" sublist (p 40).

Type: `string`

Default: `"L2 Squared"`

Penalty Parameter. The parameter is typically used when the citizen is solving a subproblem on behalf of another citizen. For more information, see the "Penalty Parameter" parameter in the "Citizen GSS-NLC" sublist (p 41).

Type: `double`

Default: `1.0`

Penalty Smoothing Value. The parameter is typically used when the citizen is solving a subproblem on behalf of another citizen. For more information, see the "Penalty Smoothing Value"

parameter in the "Citizen GSS-NLC" sublist (p 42).

Type: `double`

Default: `0.0`

Display. Specifies how much information to print about operation of the GSS citizen. Possible values are:

`0`	display nothing
`1`	display the final solution and each new "best point"
`2`	display the above, and all generated trial points
`3`	display the above, and all search directions

Type: `int`

Default: `0` (display nothing)

4.8 Citizen GSS-NLC Sublist Parameters

This sublist determines operational specifics of a Generating Search Set algorithm for problems with nonlinear constraints. The citizen is designed for problems with continuous variables, a single start point, and any set of constraints. The algorithm solves a sequence of linearly constrained subproblems using child instances of the GSS citizen (see Section 3.3 for more information about the GSS-NLC algorithm). Most parameters from the "Citizen GSS" sublist (Section 4.7) can be provided in the GSS-NLC sublist, and will be passed directly to the GSS child citizens. If a GSS parameter is listed below as a GSS-NLC parameter, then it means GSS child citizens will receive a modified version as explained below.

Type. Specifies the citizen type with a special string value.

Type: `string`

Must be: `"GSS-NLC"`

Penalty Function. Specifies the penalty function to use when computing a penalty term for any nonlinear constraints. The penalty term equals the penalty function times the current penalty parameter. The penalty term causes the objective function to become worse if constraints are violated. For instance, if the `Objective Type` parameter is `"Minimize"`, then a positive penalty term is added to the objective, and this total value is minimized by GSS searching. If the parameter is `"Maximize"`, then a positive penalty term is subtracted.

Penalty functions are identified by a string name. The notation below defines a vector $c(x)$ composed from the nonlinear constraints defined in equation 1 (p 7). The vector includes all nonlinear

equality constraints $c_{\mathcal{E}}(x)$ and all violated nonlinear inequalities from $c_{\mathcal{I}}(x)$. Possible values for the penalty function are:

`"L2 Squared"`	$\|c(x)\|_2^2$
`"L1"`	$\|c(x)\|_1$
`"L1 (smoothed)"`	smoothed version of $\|c(x)\|_1$
`"L2"`	$\|c(x)\|_2$
`"L2 (smoothed)"`	smoothed version of $\|c(x)\|_2$
`"L_inf"`	$\|c(x)\|_\infty$
`"L_inf (smoothed)"`	smoothed version of $\|c(x)\|_\infty$

Smoothed versions are described fully in [2]. The degree of smoothing is determined by the `Penalty Smoothing Value`.

Type: `string`

Default: `"L2 Squared"`

Penalty Parameter. Specifies the initial value of the penalty parameter. The penalty function is multiplied by the current penalty parameter to create a penalty term, which combines with the objective value. The penalty parameter may be updated after solving a GSS subproblem, and the new value is based on `Penalty Parameter Increase`. The value cannot be negative.

Type: `double`

Default: `1.0`

Penalty Parameter Increase. Specifies the factor by which the penalty parameter is increased. Increases happen if a GSS subproblem converges to a point that is sufficiently infeasible with respect to the nonlinear constraints. See [2] for more information. The value must be greater than one.

Type: `double`

Default: `2.0`

Penalty Parameter Maximum. Specifies the maximum value that the penalty parameter can achieve after multiplying by `Penalty Parameter Increase`. The maximum determines when the GSS-NLC citizen will "give up" trying to reach feasibility: the citizen stops if the penalty parameter is at its maximum, the current subproblem solution is infeasible, and the subproblem solution is unchanged from the previous subproblem's. The value cannot be less than the value of `Penalty Parameter`.

Type: `double`

Default: `1.0e+8` (10^8)

Penalty Smoothing Value. Specifies the initial value of the smoothing parameter α used in smoothed penalty functions. The smoothing parameter may be updated after solving a GSS subproblem, and the new value is based on `Smoothing Value Decrease`. The value must satisfy $0 \leq \alpha \leq 1$. This parameter is ignored if `Penalty Function` is not smoothed.

Type: `double`

Default: `0.0`

Smoothing Value Decrease. Specifies the factor by which the smoothing parameter is decreased. Decreases happen if a GSS subproblem converges to a point that is sufficiently infeasible with respect to the nonlinear constraints. The value must be positive and less than one. This parameter is ignored if `Penalty Function` is not smoothed.

Type: `double`

Default: `0.5`

Smoothing Value Minimum. Specifies the minimum value that the smoothing parameter can achieve after multiplying by `Smoothing Value Decrease`. The value cannot be negative, and for the `"L1 (smoothed)"` and `"L_inf (smoothed)"` penalty functions cannot be zero. This parameter is ignored if `Penalty Function` is not smoothed.

Type: `double`

Default: `1.0e-5` (10^{-5})

Final Step Tolerance. Determines the smallest steps that GSS subproblems will try before stopping, and therefore has a strong influence on accuracy of the GSS-NLC solution. Each subproblem is passed a current `Step Tolerance` (p 36) from the GSS-NLC parent citizen. The current tolerance decreases according to `Step Tolerance Decrease` until the final tolerance is reached. The value must be positive.

Type: `double`

Default: `0.001`

Initial Step Tolerance. Specifies the initial step tolerance that is passed to the first GSS subproblem as `Step Tolerance` (p 36). The value cannot be smaller than `Final Step Tolerance`.

Type: `double`

Default: `0.1`

Step Tolerance Decrease. Specifies the factor by which the step tolerance is decreased before starting a new GSS subproblem. The value cannot be negative or greater than one.

Type: `double`

Default: `0.5`

Maximum Evaluations. Specifies how many trial points should be evaluated before the GSS-NLC citizen halts. The limit applies to executed function evaluations and does not include trial points found in the Cache. The parameter limits evaluations by each instance of a GSS-NLC citizen, whether created directly from the configuration file or called as a subproblem on behalf of another citizen. A negative value means there is no limit on the number of evaluations.

Type: `int`

Default: `-1` (unlimited)

Max Subproblem Evaluations. Specifies the maximum number of trial points evaluated by each GSS subproblem. The limit applies to executed function evaluations and does not include trial points found in the Cache. The work done by a subproblem is primarily controlled by the step tolerance, but this parameter provides a separate way to limit work. A negative value means there is no limit on the number of evaluations. The value should not exceed `Maximum Evaluations` (p 43).

Type: `int`

Default: `-1` (unlimited)

Ignore Other Points. Specifies whether GSS subproblems ignore evaluated points that were generated by other citizens. If false, then subproblems will consider all evaluated points and center the search around the best point found, regardless of how that point was generated. If true, then the citizen determines a best point only from the set of candidates that it generated.

Type: `bool`

Default: `false`

Display. Specifies how much information to print about operation of the GSS-NLC citizen. Possible values are:

- `0` display nothing
- `1` display the final solution and initial parameters
- `2` display the above, and interactions with subproblems

Type: `int`

Default: `0` (display nothing)

Display Subproblem. Specifies how much information to print about operation of the GSS citizen subproblems. Possible values are:

- `0` display nothing
- `1` display the final solution and each new "best point"
- `2` display the above, and all generated trial points
- `3` display the above, and all search directions

Type: `int`

Default: `0` (display nothing)

5 Calling an Application

Every optimization application must provide a way for HOPSPACK to compute the objective function and nonlinear constraint values at a given point. HOPSPACK provides a simple and flexible method using system calls, described in Section 5.1. The user can also modify HOPSPACK source code to evaluate points in some other manner, as discussed in Section 5.2.

5.1 Evaluation by System Call

The default mechanism in HOPSPACK is to call an external program for function evaluations. Generally, the user writes a simple wrapper that calls the true application. Variable values are passed in a short text file and function results are passed back in a separate text file. HOPSPACK writes the variable values, makes a system call from C++ to execute the application, waits for it to complete, and reads the output file. This simple mechanism provides maximum flexibility for the application. It can be written in any language (Fortran, C, C++, Perl, MATLAB, etc.), consist of multiple executables strung together (e.g., using a shell or .bat script), and reference external data sources. The application itself can use MPI to run in parallel, although HOPSPACK will not adjust its load balancing (you must figure how to allocate nodes between HOPSPACK and the application copies). The same application wrapper will work with the MPI, multithreaded, and serial versions of HOPSPACK.

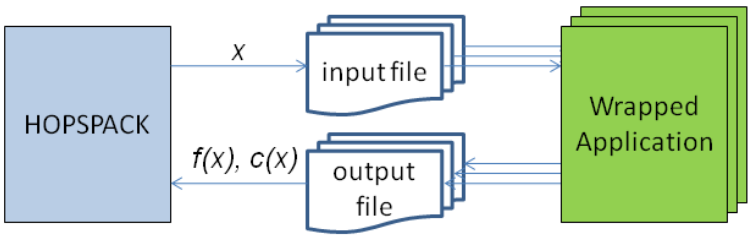


Figure 5. HOPSPACK communication with an application.

Figure 5 shows the flow of data from HOPSPACK to an application. Specific variable values at a trial point x are written to an input file, and the evaluated objective $f(x)$ plus any nonlinear constraints $c(x)$ are returned via an output file. The figure shows multiple instances of the application running in parallel. Each instance has its own input and output files, and must run independent from other application instances. Figures 2 (MPI) and 3 (MT) show examples of how the applications and HOPSPACK might be distributed across processors and threads. HOPSPACK helps maintain independence of parallel application instances by defining a unique "tag number" for each evaluation point. The tag is used to form unique input and output file names and is made available to the application instance. Input and output file names are generated by HOPSPACK as the concatenation of a string defined by parameter `Input Prefix` (p 30) or `Output Prefix` (p 31), the tag number, and the evaluation type (discussed below). The unique tag number allows all files to coexist in the same directory. If the application creates temporary files, it should use the tag number to name the files so there is no conflict between parallel instances. Tag numbers are generated from a counter that is reset whenever HOPSPACK starts; hence, tag numbers can

be reused when HOPSPACK restarts.

Input and output file information depends on the type of evaluation information desired, which is one of the following:

`F`	Evaluate the objective $f(x)$ only (no constraints)
`C`	Evaluate nonlinear constraints $c(x)$ only (no objective)
`FC`	Evaluate $f(x)$ and nonlinear constraints $c(x)$

The default type is `FC` since the GSS solver requires all information at each trial point. However, a different solver might separate processing of the objective and constraints; for example, a solver may want to first find a completely feasible point before requesting the objective value.

The input file begins with two lines of header information and then the value of each variable. The first line contains the evaluation type, and the second line gives the number of variables. This is followed by variable values, one per line. The text below shows an example of an input file generated by HOPSPACK for a problem with two variables:

```
FC
2
-1.50000000000000e+00
3.91875000000000e+00
```

The output file format is similar. Objective values are written first (assuming the evaluation type is `F` or `FC`), and then any nonlinear constraints (for type `C` or `FC`). Constraints are written as two vectors of values: one for equalities and one for inequalities. Objective values are also written as a vector of values, allowing applications with multiple objectives. In all three cases the format of a vector begins with a header line giving the number of items, and then a list of values, one per line. If a function cannot be evaluated, the string "`DNE`" should be returned, indicating a value does not exist at the trial point. The text below shows an example of an output file for type `FC`. There is one objective (with $f(x) = 5$), no equality constraints, and two inequalities. Nonlinear inequalities follow a "greater than zero" convention (Section 4.1), so this particular point is feasible with respect to the first inequality and infeasible with respect to the second. Note that HOPSPACK expects the number of objectives to match the value of configuration parameter `Number Objectives` (p 25), and the number of constraints to match `Number Nonlinear Eqs` (p 26) and `Number Nonlinear Ineqs` (p 26).

```
1
5.00000000000000e+00
0
2
1.25000000000000e+02
-2.08164062500000e+00
```

When the application wrapper is called to evaluate a particular trial point, it is given four command line arguments: the input file name, output file name, tag number, and evaluation type. The last argument is repeated on the first line of the input file. The application wrapper must read a HOPSPACK input file and create a new output file before completing. The wrapper executable itself should return zero if successful; any other value indicates to HOPSPACK that the evaluation

failed. An example written in C is provided in the file examples/1-var-bnds-only/var_bnds_only.c. Examples written in C++ are provided in similar subdirectories, including a problem with nonlinear constraints in examples/4-nonlinear-constraints/nonlinear_constraints.cpp. The HOPSPACK source code that calls the application is located in src/src-evaluator/HOPSPACK_SystemCall.cpp.

The application wrapper can return a short error message that will be reported in HOPSPACK output and passed to solvers. The point will be marked as unevaluated, causing it to be ignored by most solvers. The error message can be a useful mechanism for reporting types of evaluation failures, instead of simply failing. To send an error message the wrapper executable should return zero (as though it succeeded), and the message should be on the first line of the output file instead of the number of objectives. Everything after the message will be ignored. HOPSPACK will attach the message to the point and notify all solvers that the point could not be evaluated.

5.2 Linking Evaluation Code

The default HOPSPACK mechanism described in Section 5.1 evaluates functions by calling the application as a separate process. A user familiar with C++ can instead link HOPSPACK with the application code and call it directly. This mechanism eliminates separate application processes and file-based communications. Direct calls will therefore run faster, although the time savings may be negligible compared to function evaluation times. Direct calls also allow run time control of the application through customization of the "Evaluator" sublist of configuration parameters.

Calling the application directly requires modification of HOPSPACK source code, and linking generally requires changing a CMake build file. There are different ways to accomplish this. One option is provided in the directory examples/linked-evaluator-example. This example replaces the `HOPSPACK::EvaluatorDefault` class with a custom `ExampleLinkedEvaluator` class. The new class implements the interface `HOPSPACK::Evaluator`, which includes methods `evalF()` and `evalFC()` for invoking the application. A one-line source code change is needed to construct an instance of the new evaluator instead of `EvaluatorDefault`. More information is given in the file examples/linked-evaluator-example/README_linked_evaluator.txt.

Another option is to define a new value for the parameter `Evaluator Type` (p 30) in addition to the default value `"System Call"`. The new value should be added to method `newInstance()` of the `HOPSPACK::EvaluatorDefault` class, located in file src/src-evaluator/HOPSPACK_EvaluatorDefault. The custom evaluator must subclass `HOPSPACK::EvaluatorDefault`. An advantage of this approach is that the evaluator type can be altered at run time through the configuration file.

If the application depends on specific libraries, then their names must be given to CMake. See Section 8.4 for details.

An application called directly has the potential to crash HOPSPACK. Be sure to surround the call with `try/catch` handlers to trap any application errors. If an error is found, the custom evaluator should return `DNE` values for the functions and a short error message that will be reported in HOPSPACK output.

An application called directly from the multithreaded version of HOPSPACK must be thread safe. HOPSPACK may call `evalFC()` simultaneously from many threads in the same shared memory space.

5.3 Evaluation Tips

For best results, applications should use full numerical precision when passing values. HOPSPACK writes variable values to the input file with 15 significant digits, the maximum precision for double precision on most 32-bit machines. The number of digits can be modified with the configuration parameter `File Precision` (p 31). The application wrapper should write results to the output file using the most precision possible.

An application with nonlinear constraints might benefit from careful ordering of calculations in the application wrapper. For instance, if some constraints cannot be violated (sometimes called "hard constraints"), then it may be best to check this first and not try to compute the objective function at an infeasible point. In this case it is appropriate to return `DNE` for any unevaluated functions. If the objective is well defined but expensive to compute, then it may be best to skip evaluating at an infeasible point, returning `DNE` for the objective. This causes the constraint to appear as an infinite barrier to solvers, because an objective value of `DNE` is treated as a value of infinity ($+\infty$ if minimizing, $-\infty$ if maximizing). An infinite barrier is nonsmooth and could slow convergence, but the overall time savings might be worthwhile.

Some solvers might not be capable of working with nonlinear constraints. In this case the evaluation might choose to return `DNE` when a point is infeasible. However, this is not recommended if using the default GSS solver in HOPSPACK. GSS computes a penalty for infeasible points and can make use of the extent to which a point violates constraints. See Section 3.3 for more.

6 Building HOPSPACK

HOPSPACK is written with the intent of allowing user modifications and extensions. All code is written in C++. HOPSPACK uses the CMake build system (http://cmake.org/) to support compilation on multiple platforms, including Linux, Windows, and Mac OSX. This section describes the process of installing source code, third party libraries, and building HOPSPACK executables. Section 7 provides examples of modifying or extending the source code.

Several steps are required to build HOPSPACK. A quick outline is below, and full details for various platforms follow.

6.1 Download HOPSPACK Source Code

Follow the links at

https://software.sandia.gov/trac/hopspack

to find the download page. Please register your email address with accurate and complete information. We ask for this information as a courtesy in exchange for our free software. Having accurate user data allows us to better ascertain in what way HOPSPACK is used, which may influence future development. Your email address will remain strictly confidential and will not be used unless you request to be on the HOPSPACK Users Mailing List. Remember the email address you register so you can registration the next time.

Download the source code. For convenience, it is supplied in both Windows compressed file form and Unix compressed tar file form. The contents are the same.

Save the compressed file to any directory and unzip it. You should see a directory structure like the following:

```
hopspack-2.0-src
    doc
    examples
    src
    test
```

6.2 Download and Install CMake

CMake is a leading open-source build system that supports multiple operating systems. You need to download a CMake binary distribution (typically, 5-10 Mbytes in size) appropriate for your

operating system and install it. If HOPSPACK will run on different machines, then install CMake on each target machine. The installation creates a CMake tool that will be used to construct platform-specific build scripts for compiling HOPSPACK source code.

Visit http://cmake.org/ and find a recent release of CMake for your target operating system. The CMake release must be 2.6.2 or later. At the time this documentation was produced, the CMake distribution could be found by clicking on RESOURCES and then Download to reach http://cmake.org/cmake/resources/software.html. Only the binary distribution is needed (no CMake source code). For example, `cmake-2.6.3-Linux-i386.tar.gz` was the file name for an x86 Linux machine, and `cmake-2.6.3-win32-x86.exe` the file name for an x86 Windows machine.

Installation of CMake is very simple, and explained on the CMake download page. For example, on a Linux machine just unpack the file to any directory (this procedure does not require root privileges). It should create a new subdirectory tree with a name like `cmake-2.6.3-Linux-i386`. Just add the subdirectory `cmake-2.6.3-Linux-i386/bin` to `PATH`.

On Windows, run the CMake distribution file to start an installation wizard and follow the directions. By default, CMake will install at `C:\\Program Files\CMake 2.6` and create a Start Menu entry that invokes the CMake GUI interface. If you prefer to run the command line version of CMake, then click a wizard button that adds CMake to `PATH`.

6.3 Build an LAPACK Library

A third party LAPACK (Linear Algebra PACKage) library is required for optimization problems with general linear constraints. Simple variable bounds do not require LAPACK. If your problems do not have general linear constraints, then skip the rest of this section, but add the command line option `-Dlapack:BOOL=false` when building HOPSPACK. The option tells the build to modify source code so that no calls to LAPACK are made; however, it also prevents HOPSPACK from solving problems with linear constraints.

Your system may already have LAPACK installed. For instance, on some Linux distributions LAPACK is available in the file /usr/lib/liblapack.a. In this case CMake should find it automatically and no further effort is needed. Try building the serial executable as described in Section 6.4; the CMake configuration will tell you clearly whether an LAPACK library was found.

If LAPACK was not found on your system, or you prefer a particular version, then the library must be installed. LAPACK libraries are available from many sources. Perhaps the most common version is from Netlib, freely available at http://netlib.sandia.gov/lapack. Other possibilities are vendor-provided libraries like the Intel MKL or AMD ACML, and tunable versions such as ATLAS.

LAPACK functions called by HOPSPACK are the following:

```
ddot       dgemv
dgelqf     dgesvd
dgemm      dgglse
```

Make sure the library contains these functions and their dependents, or there will be unresolved symbols when linking the final HOPSPACK executable. CMake will test for the presence of these functions when it configures HOPSPACK, and will halt with a warning message if it detects a problem.

Linux example of building Netlib. This example shows a particular case of building a Netlib version using the GNU compilers. Netlib produces two library files, one for BLAS functions such as `ddot` and one for LAPACK functions such as `dgglse`. The Netlib libraries are created using a Fortran compiler, so the HOPSPACK C++ executables must include a Fortran-to-C library (the CMake build process will try to do this automatically). The example also shows how to edit the HOPSPACK CMake configuration file to find the libraries if they are produced in a nonstandard location. Please note this is just one possible example and your build procedure may differ.

- Download lapack-3.1.1.tgz from http://netlib.sandia.gov/lapack
- Unpack the distribution (this example assumes the directory `/tmp` is used)
- Consult README and INSTALL/lawn81.pdf for Netlib instructions.
- For a Linux RHEL 4 machine build a minimal LAPACK as follows:
  `> cp make.inc.example make.inc`
  Edit Makefile:
    Change comments to enable building `blaslib` and `lapacklib`
    Change "`$(MAKE)`" to "`$(MAKE) double`" to avoid unnecessary objects
  `> make lib` (should produce files blas_LINUX.a and lapack_LINUX.a)
  Netlib libraries must be renamed to conform with Linux convention:
  `> mv blas_LINUX.a libblas_LINUX.a`
  `> mv lapack_LINUX.a liblapack_LINUX.a`
- Either add these options to the CMake command line:
  `-DLAPACK_LIBS="$LH/liblapack_LINUX.a;$LH/libblas_LINUX.a"`
  (where `$LH` is the LAPACK home `/tmp/lapack-3.1.1`)
- Or edit ConfigureLapack.cmake in the HOPSPACK directory:
  Find the section beginning with the message "Linear constraints allowed"
  Comment out option 1 and uncomment the lines for option 3
  Change the path in option 3 to the location of your libraries
  (Use option 2 if your libraries are in a standard location)
- If CMake has trouble finding the Fortran-to-C library, try adding:
  `-DLAPACK_ADD_LIBS="gfortran"`

Windows example of using Netlib with MSVC. This example uses a precompiled Netlib distribution made with the Microsoft Visual C++ compiler (MSVC). Netlib code is traditionally written in Fortran, but it is often more convenient to use the free MSVC compiler on Windows. Netlib provides a version of the source code that is translated to C, called CLAPACK. A single precompiled library file includes BLAS and LAPACK functions. Symbol names are different, but HOPSPACK works with CMake to recognize this and link correctly. The procedure follows:

- Download CLAPACK3-Windows.zip from http://netlib.sandia.gov/clapack
- Unzip the distribution in any directory; here, assume `c:\temp` is used
- Either add this option to the CMake command line:
  `-DLAPACK_LIBS=c:\temp\CLAPACK3-Windows\CLAPACK\Release\clapack.lib`
- Or edit ConfigureLapack.cmake in the HOPSPACK directory:
  Find the section beginning with the message "Linear constraints allowed"
  Comment out option 1 and uncomment the lines for option 4
  Change the path in option 4 to the location of your libraries

Linux example of using Intel MKL. The Intel MKL contains routines for LAPACK and many other math functions that are specially tuned for Intel microprocessors. You could use the MKL builder tool to create a single library containing just the LAPACK routines needed by HOPSPACK. In that case, pass the library to CMake using the command line option `-DLAPACK_LIBS`. Another technique is to provide all the appropriate MKL libraries to CMake and let the linker find what it needs. Assuming `$MKL_LIB` is the directory where MKL libraries are stored, the CMake command line options are (for MKL version 9.0):

```
-DLAPACK_LIBS="$MKL_LIB/libmkl_lapack.a;$MKL_LIB/libmkl_ia32.a"

-DLAPACK_ADD_LIBS=$MKL_LIB/libguide.so
```

6.4 Build and Test the "serial" HOPSPACK Executable

The CMake tool constructs platform-specific build scripts for compiling and linking executables. We recommend making an "out of source" build, instead of building the object and executable files in the source directories. This is easy to do with CMake and allows the existence of multiple builds without conflict; for instance, a serial and MPI build.

To create an out of source build, make a clean directory, change to it, and run CMake from this directory. CMake allows the build directories to be anywhere, but in the remainder of this section we assume a clean directory is created at the same level as hopspack-2.0-src. After building all versions, the directory structure will look like the following:

hopspack-2.0-src	
doc	(provided)
examples	(provided)
src	(provided)
test	(provided)
build_serial	(create this and build in it)
HOPSPACK_main_serial	(built by CMake)
examples	(built by CMake)
1-var-bnds-only	(built by CMake)
...	(built by CMake)
test	(built by CMake)
build_mpi	(see Section 6.6)
build_mt	(see Section 6.5)

The examples below show how to run CMake on various platforms. For trouble-shooting or customizing CMake, see Section 8. For information on linking with an LAPACK library, see Section 6.3.

Linux (build serial HOPSPACK). Start at the HOPSPACK parent directory and run the following commands:

```
> mkdir build_serial
> cd build_serial
> cmake ../hopspack-2.0-src
```

```
-- The CXX compiler identification is GNU
-- ...
-- Build files have been written to:  ...
> make
> make test
```

The execution of `cmake` displays several lines of informational output, only a few of which are shown above. Its behavior is roughly similar to a Unix "autoconf" or "config" tool. It produces a subdirectory structure similar to that of hopspack-2.0-src, with `Makefile` files that work with the chosen compiler. Running `make` in the last step produces the HOPSPACK executable, test executables, and optimization evaluators in examples. Running `make test` invokes CTest (a CMake utility) to run any automated tests that come with HOPSPACK. They should all pass.

Now change to the examples directory. There should be a number of subdirectories named 1-var-bnds-only, 2-linear-constraints, etc., and a README.txt file. Each subdirectory contains HOPSPACK configuration parameters and a small executable that evaluates optimization objectives and constraints at a given point. Run the first example:

```
> cd examples/1-var-bnds-only
> ../../HOPSPACK_main_serial example1_params.txt
```

As explained in the README.txt file, this solves a simple two-dimensional minimization problem with bound constraints. Check the solution against the value listed in README.txt. Try 2-linear-constraints if your configuration includes an LAPACK file for problems with linear constraints.

A common problem on Linux machines is failure of example evaluations because the current directory is not in the `PATH` environment variable. An error message

```
ERROR: Call failed:  'var_bnds_only ...'  <SystemCall>
```

means the Evaluator in HOPSPACK could not run the example executable. An easy fix is to add the current directory to the PATH environment variable:

```
> export PATH=$PATH:.
```

Alternatively, edit example1_params.txt and change the parameter defining the executable to be

```
"Executable Name" string "./var_bnds_only"
```

Mac OSX (build serial HOPSPACK). This section assumes a recent version of XCode is installed on the Mac, providing a g++ compiler and LAPACK libraries. For example, *XCode 3.1.2* on Mac OSX 10.5.8 provides the files /usr/lib/libblas.dylib and /usr/lib/liblapack.dylib.

Open a Mac Terminal Window and follow the same procedure as the Linux build example described immediately above. The only difference is that the additional LAPACK library name must be passed to CMake during configuration. Do this with the option `-DLAPACK_ADD_LIBS`:

```
> mkdir build_serial
> cd build_serial
> cmake ../hopspack-2.0-src -DLAPACK_ADD_LIBS=/usr/lib/libblas.dylib
-- ...
> make (or gnumake)
```

This will build the executable program `HOPSPACK_main_serial`.

Now change to the examples directory. There should be a number of subdirectories named 1-var-bnds-only, 2-linear-constraints, etc., and a README.txt file. Each subdirectory contains HOPSPACK configuration parameters and a small executable that evaluates optimization objectives and constraints at a given point. Run the first example:

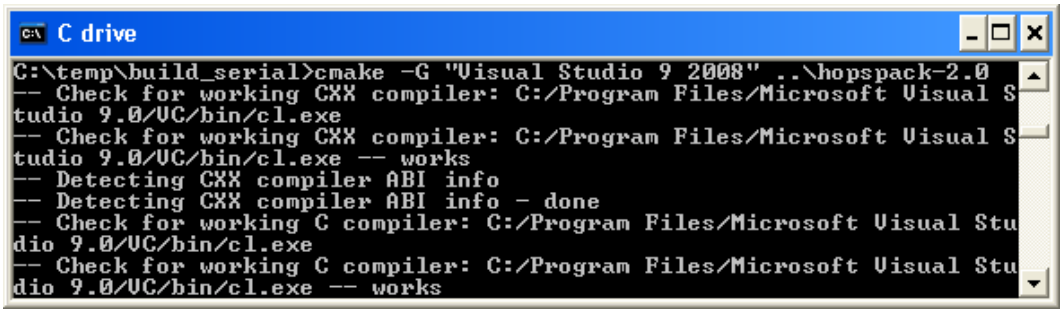
```
> cd examples/1-var-bnds-only
> ../../HOPSPACK_main_serial example1_params.txt
```

As explained in the README.txt file, this solves a simple two-dimensional minimization problem with bound constraints. Check the solution against the value listed in README.txt.

A common problem is failure of example evaluations because the current directory is not in the `PATH` environment variable. See the Linux build example above for more information.

Windows using Visual Studio (build serial HOPSPACK). CMake can generate a Microsoft Visual Studio project for the HOPSPACK source code. At the time this documentation was produced, CMake was unable to create Visual Studio project files from a free Visual Studio Express Edition (there is a known bug in locating the compiler). Hence, this section assumes a full Visual Studio product is installed (for example, *Microsoft Visual Studio 2008* with the version 9.0 C++ compiler). A subsequent example describes how CMake can produce a set of `Makefile` files that work with the command line `nmake` tool in the Express Edition.

CMake can execute in a GUI or from the command line. This example uses a Windows DOS-like command line console such as the one below.



First, make sure environment variables are configured for the Microsoft compiler. If installed in its default location, this is accomplished (for version 9.0) by running:

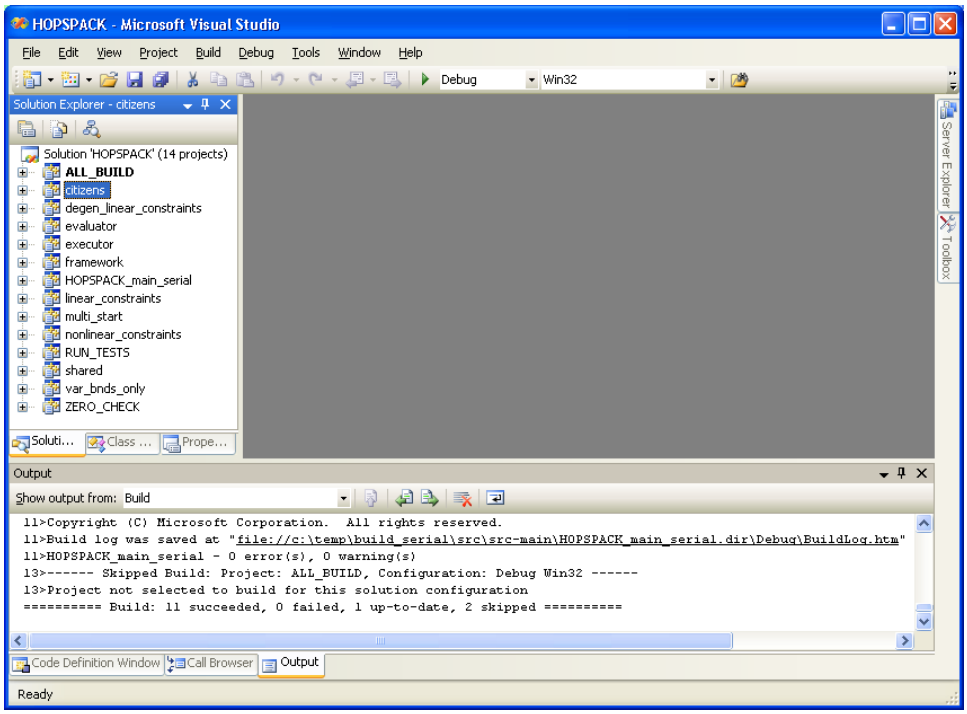
```
> c:\Program Files\Microsoft Visual Studio 9.0\VC\vcvarsall.bat
```

Start at the directory where the HOPSPACK parent directory exists and run the following commands:

```
> mkdir build_serial
> cd build_serial
> cmake -G "Visual Studio 9 2008" ..\hopspack-2.0-src
-- Check for working CXX compiler:  C:\Program Files ...
-- ...
-- Build files have been written to:  ...
```

The execution of `cmake` displays several lines of informational output, only a few of which are shown above. It produces a subdirectory structure similar to that of hopspack-2.0-src, and a file ALL_BUILDS.vcproj with the main Visual Studio project.

Start Visual Studio and open the file ALL_BUILDS.vcproj. It contains projects for all the libraries and executables built by HOPSPACK. If you build `ALL_BUILDS` then Visual Studio will compile and link everything, including the serial HOPSPACK executable, examples, and tests. A successful build is shown in the screen shot below.



You must return to a command line console to run HOPSPACK. The main executable should be in the src directory: build_serial\src\HOPSPACK_main_serial.exe.

Now change to the examples directory. There should be a number of subdirectories named 1-var-bnds-only, 2-linear-constraints, etc., and a README.txt file. Each subdirectory contains HOPSPACK configuration parameters and a small executable that evaluates optimization objectives and constraints at a given point. You may have to edit the parameters file in each example and tell it where the optimization executable is located. Run the first example:

```
> cd examples\1-bnd-vars-only
```

Edit example1_params.txt and change the "Executable Name" parameter to be

```
"Executable Name" string "Debug\var_bnds_only.exe"
```

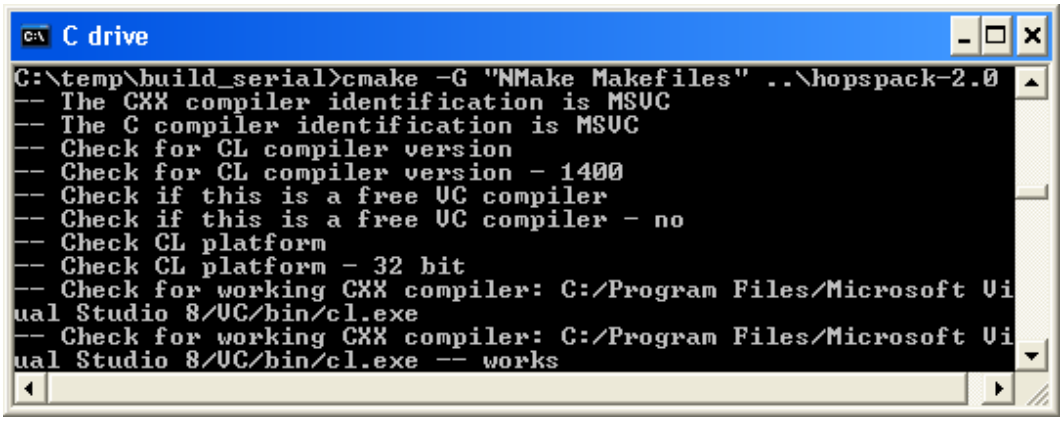
```
> ..\..\HOPSPACK_main_serial.exe example1_params.txt
```

As explained in the README.txt file, this solves a simple two-dimensional minimization problem with bound constraints. Check the solution against the value listed in README.txt. Try 2-linear-constraints if your configuration includes an LAPACK file for problems with linear constraints.

Windows using NMake (build serial HOPSPACK). CMake can generate a set of `Makefile` files that work with the Visual Studio command line `nmake` tool. The `nmake` facility is provided

with the full Microsoft Visual Studio product or the free Visual Studio Express Edition (available from Microsoft).

This section assumes *Visual C++ 2005 Express Edition* with the version 8.0 compiler is installed. All commands are run from a Windows DOS-like command line console such as the one below.



First, make sure environment variables are configured for the Microsoft compiler. If installed in its default location, this is accomplished (for version 8.0) by running:

```
> c:\Program Files\Microsoft Visual Studio 8\Common7\Tools\vsvars32.bat
> c:\Program Files\Microsoft Platform SDK...\SetEnv.cmd
```

Start at the directory where the HOPSPACK parent directory exists and run the following commands:

```
> mkdir build_serial
> cd build_serial
> cmake -G "NMake Makefiles" ..\hopspack-2.0-src
-- The CXX compiler identification is MSVC
-- ...
-- Build files have been written to:  ...
> nmake
```

The execution of `cmake` displays several lines of informational output, only a few of which are shown above. It produces a subdirectory structure similar to that of hopspack-2.0-src, with `Makefile` files that work with the chosen compiler. Running `nmake` in the last step produces the HOPSPACK executable, test executables, and optimization evaluators in examples.

Now change to the examples directory. There should be a number of subdirectories named 1-var-bnds-only, 2-linear-constraints, etc., and a README.txt file. Each subdirectory contains HOPSPACK configuration parameters and a small executable that evaluates optimization objectives and constraints at a given point. Run the first example:

```
> cd examples\1-bnd-vars-only
> ..\..\HOPSPACK_main_serial.exe example1_params.txt
```

As explained in the README.txt file, this solves a simple two-dimensional minimization problem with bound constraints. Check the solution against the value listed in README.txt. Try 2-linear-constraints if your configuration includes an LAPACK file for problems with linear constraints.

6.5 Build and Test an "mt" HOPSPACK Executable

This section assumes you are making an "out of source" multithreaded build in a separate directory from the serial build of Section 6.4. After building, the directory structure will look like the following:

hopspack-2.0-src	
doc	(provided)
examples	(provided)
src	(provided)
test	(provided)
build_serial	(see Section 6.4)
build_mt	(create this and build in it)
HOPSPACK_main_threaded	(built by CMake)
examples	(built by CMake)
1-var-bnds-only	(built by CMake)
2-linear-constraints	(built by CMake)
test	(built by CMake)

The build procedure is almost identical to that of Section 6.4. An extra option is passed to CMake that instructs it to find multithreading libraries and compile additional thread-based source files. You can create separate serial and MPI versions of HOPSPACK if for some reason multithreading libraries are not available on your machine.

All the examples in Section 6.4 begin with three instructions: create a new directory, change to it, and run CMake. To build the multithreaded version, similar instructions are typed in at the command line, but with the option `-Dmt=yes`:

```
> mkdir build_mt
> cd build_mt
> cmake ../hopspack-2.0-src -Dmt=yes
```

From this point, the build procedure is identical to Section 6.4. Note that CMake accepts any of the option values `-Dmt=yes`, `-Dmt=true`, or `-Dmt=on`.

Assuming the build completed successfully, change to the examples directory. There should be a number of subdirectories named 1-var-bnds-only, 2-linear-constraints, etc., and a README.txt file. Each subdirectory contains HOPSPACK configuration parameters and a small executable that evaluates optimization objectives and constraints at a given point. Run the first example:

```
> cd examples/1-var-bnds-only
> ../../HOPSPACK_main_threaded example1_params.txt
```

As explained in the README.txt file, this solves a simple two-dimensional minimization problem with bound constraints. Check the solution against the value listed in README.txt. The `Number Threads` parameter in the "Mediator" sublist (p 32) determines how many threads are created. Try increasing the number of threads and observe that more "Eval workers" are used.

If the `Display` parameter in the "Mediator" sublist (p 35) is 3 or larger, then a timing report for each evaluation worker will be printed after HOPSPACK completes. The value displayed is

the cumulative wall clock time that the Executor believes a worker is busy. This should be nearly the same as the time consumed by evaluations on a worker, assuming the machine has sufficient processors to handle each worker thread. However, if workers do not have available processors when they are assigned work by the Executor, then the displayed time will be longer than actual evaluation time.

Source files used in the multithreaded version are identical to those used in the serial version except for the main routine (HOPSPACK_main_threaded.cpp versus HOPSPACK_main_serial.cpp), and the executor (HOPSPACK_ExecutorMultiThreaded.cpp versus HOPSPACK_ExecutorSerial.cpp). In addition, the "shared" library includes classes that wrap native threading functions (src/src-shared/HOPSPACK_Thread*).

6.6 Build and Test an "mpi" HOPSPACK Executable

This section assumes you have read about building a serial executable in Section 6.4. Differences for building with MPI are discussed here, but refer to Section 6.4 for more details about building with CMake.

This section assumes you are making an "out of source" MPI build in a separate directory from the serial build of Section 6.4. After building, the directory structure will look like the following:

hopspack-2.0-src	
doc	(provided)
examples	(provided)
src	(provided)
test	(provided)
build_serial	(see Section 6.4)
build_mpi	(create this and build in it)
HOPSPACK_main_mpi	(built by CMake)
examples	(built by CMake)
1-var-bnds-only	(built by CMake)
2-linear-constraints	(built by CMake)
test	(built by CMake)

The build procedure is almost identical to that of Section 6.4. An extra option is passed to CMake that instructs it to find MPI libraries and compile with an MPI-aware compiler. You can create separate serial and multithreaded versions of HOPSPACK if for some reason MPI fails to build on your machine.

All the examples in Section 6.4 begin with three instructions: create a new directory, change to it, and run CMake. To build the MPI version, similar instructions are typed in at the command line, but with the option `-Dmpi=yes`:

```
> mkdir build_mpi
> cd build_mpi
> cmake ../hopspack-2.0-src -Dmpi=yes
```

If CMake has trouble finding your MPI-aware compiler, try specifying it as a command line parameter; for example:

```
> cmake ../hopspack-2.0-src -Dmpi=yes -DMPI_COMPILER=/tmp/mpich-1.2.7p1/bin/mpicxx
```

If this fails, consider modifying ConfigureMPI.cmake. This file contains some comments about the procedure CMake uses to find and configure MPI.

From this point, the build procedure is identical to Section 6.4. Note that CMake accepts any of the option values `-Dmpi=yes`, `-Dmpi=true`, or `-Dmpi=on`.

Assuming the build completed successfully, change to the examples directory. There should be a number of subdirectories named 1-var-bnds-only, 2-linear-constraints, etc., and a README.txt file. Each subdirectory contains HOPSPACK configuration parameters and a small executable that evaluates optimization objectives and constraints at a given point. Run the first example:

```
> cd examples/1-var-bnds-only
> mpirun -np 2 ../../HOPSPACK_main_mpi example1_params.txt
```

As explained in the README.txt file, this solves a simple two-dimensional minimization problem with bound constraints. Check the solution against the value listed in README.txt. The `Number Processors` parameter in the "Mediator" sublist (p 32) determines how many processors are used. Try increasing the number (along with the argument for `-np`) and observe that more "Eval workers" are used.

If the `Display` parameter in the "Mediator" sublist (p 35) is 3 or larger, then a timing report for each evaluation worker will be printed after HOPSPACK completes. The value displayed is the cumulative wall clock time that the Executor believes a worker is busy. Time is measured by the Executor on the main MPI node, not the worker. The Executor measures from the moment an MPI message is sent to a worker to the moment an MPI reply is received.

Source files used in the MPI version are identical to those used in the serial version except for the main routine (HOPSPACK_main_mpi.cpp versus HOPSPACK_main_serial.cpp), and the executor (HOPSPACK_ExecutorMpi.cpp versus HOPSPACK_ExecutorSerial.cpp). In addition, the "shared" library includes a class that wraps MPI functions (src/src-shared/HOPSPACK_GenProcComm).

7 Extending HOPSPACK

The HOPSPACK framework is written with the intention that users will extend it to suit their own needs. Software is written in C++ and follows object oriented design practices. The code compiles on major platforms using the CMake build tool (see Section 8).

Comments throughout the code conform to Doxygen standards (http://www.doxygen.org/) for automated source code documentation. HTML documentation pages generated from Doxygen are provided with the source code distribution. Open the file src/doc_doxygen/html/index.html in a browser to get started and use this documentation to learn how the software is layed out. New documentation can be generated for modified code if you install the Doxygen tool on your machine and run it using the configuration file src/Doxyfile.txt.

A common extension is to call an application directly from the HOPSPACK evaluator, rather than start it as a separate process for every trial point. This extension is described in Section 5.2.

HOPSPACK can be embedded as a callable library, provided the parent application is careful in devoting the proper number of parallel resources to HOPSPACK. User code needs to construct an instance of the `Executor` and `Hopspack` classes, and form a `ParameterList` of configuration parameters. Then the code simply calls the method `Hopspack::solve()`. Refer to src/src-main/HOPSPACK_main_serial.cpp and src/src-main/HOPSPACK_main_threaded.cpp for examples.

7.1 Writing a New Citizen

Users are encouraged to write their own citizen code to test out new algorithm ideas or to create hybrid solution methods. The HOPSPACK 2.0 release provides only three citizens (GSS, GSS-NLC, GSS-MS), and the multi-start citizen is just a placeholder for more sophisticated algorithms. New citizens can be added to the source code alongside the existing citizens. Applications can enable or disable citizens in any combination through the configuration file.

A new citizen must implement a subclass of `Citizen`, which is declared in the file src/src-citizens/HOPSPACK_Citizen.hpp. It is best to create a new subdirectory under src/src-citizens for each new citizen. A new citizen should define a unique string as the value of the `Type` parameter in sublist "Citizen"; for example, the GSS citizen uses the value `"GSS"` (p 36). Then code should be added to the `Citizen::newInstance()` method of src/src-citizens/HOPSPACK_Citizen.cpp, using the string identifier. This code should recognize the citizen type and construct an instance of the subclass.

A citizen must implement all of the virtual void methods in `Citizen`. Most of these are fairly simple and can be written by looking at the GSS and GSS-NLC citizens as examples. The heart of a citizen's activity is the method `exchange()`. This is called once per iteration by the Mediator. As described in Section 3.1, the citizen receives a list of newly evaluated trial points and is expected to return a new list of candidate points. In effect, the method exchanges new trial points for old ones. The citizen receives points from all other citizens, but is supplied with a corresponding list of "owner tags" so it can identify its own points. Beyond these simple requirements, a citizen is free to pursue any algorithmic method for processing old points and generating new ones. The Mediator will call `getState()` to learn if the citizen is finished.

A citizen can define its own configuration parameters to get runtime information from the user. A citizen can call subproblem solvers or be called as a subproblem (see src/src-citizens/citizen-gss-nlc for an example).

Notes in this section are admittedly brief and will be expanded in the future. Our hope is to see more citizens added to the base source code of HOPSPACK as the project evolves.

8 More About CMake

Documentation for CMake is part of the CMake installation (Section 6.2), and can be found on the CMake web site (http://cmake.org/). Files in the HOPSPACK source distribution named CMakeLists.txt or files that end with the suffix .cmake were written for HOPSPACK. Any of these CMake files can be examined and potentially edited to alter CMake behavior. The remainder of this section describes specific situations where you might want to alter behavior when building HOPSPACK.

8.1 Debugging the Build Process

Sometimes it helps to see more makefile output during compilation. On makefile systems detailed output is enabled by editing ConfigureBuildType.cmake and uncommenting the line

```
SET (CMAKE_VERBOSE_MAKEFILE ON)
```

Then you should call `cmake` in a clean "out of source" directory to rebuild the HOPSPACK makefiles.

8.2 Building a Debug Version of the Code

To compile a HOPSPACK executable with debugging symbols, use the command line option `-Ddebug:BOOL=true`. For example, on a Linux machine start in a clean "out of source" directory and call:

```
> cmake ../hopspack-2.0  -Ddebug:BOOL=true
```

Then, of course, all files must be recompiled.

8.3 Specifying a Different Compiler

Early in its configuration process, CMake chooses a C++ compiler to use. The command line version usually prints messages about its choice; for example, here is some of the output from the build of a serial HOPSPACK executable on Linux:

```
-- The CXX compiler identification is GNU
-- The C compiler identification is GNU
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
...
```

You can force CMake to use a different compiler by altering the environment variables `CXX` and `CC`. In addition, you can add compiler flags by setting `CXXFLAGS` and tell the linker to include certain libraries by setting `CMAKE_EXE_LINKER_FLAGS`. As an example, suppose the Intel C++ compiler (version 8.1) is installed on a Linux machine. Assume the bin directory containing the compiler

icc is in PATH, and that the libraries directory is placed in LD_LIBRARY_PATH. Then you instruct CMake to build a makefile system as follows:

```
> mkdir build_serial
> cd build_serial
> export CXX=icc
> export CC=icc
> export CXXFLAGS=-cxxlib-icc
> cmake ../hopspack-2.0 \
          -DCMAKE_EXE_LINKER_FLAGS="-lcprts -lcxa -lunwind"
-- The CXX compiler identification is Intel
-- The C compiler identification is Intel
...
```

8.4 Adding Libraries to an Executable

If source code modifications introduce dependencies on external libraries, then CMake must be given the library names so they can be linked with the executables.

A simple way is to add the library name explicitly in the CMake configuration file that generates an executable. For example, suppose the serial version of HOPSPACK on a Linux machine needs to link with the dl system library (perhaps the function `dlopen()` was called in a custom evaluator such as the one described in Section 5.2). A simple fix is to edit src/src-main/CMakeLists.txt and add `-ldl` in the list of `TARGET_LINK_LIBRARIES` at the bottom of the file. Assuming the library is in the system's load path, CMake will find it the next time the executable is built.

The simple fix described above is hard-coded for Linux. If the library exists on all platforms, then CMake has a better way. For example, suppose you want to link a personal library of utility functions named "myutils". On Linux this would typically be named libmyutils.a or libmyutils.so, while Windows would typically name it myutils.dll. CMake provides a utility that finds the platform-specific name:

```
FIND LIBRARY (MY_UTILS_VAR NAMES myutils DOC "find myutils")
```

This stores the platform-specific name in the CMake variable named `MY_UTILS_VAR`. Add the variable to the list of `TARGET_LINK_LIBRARIES` instead of a hard-coded name.

For more examples, look at ConfigureLapack.cmake and ConfigureSysLibs.cmake in the top directory of the HOPSPACK distribution.

References

[1] G. A. Gray and T. G. Kolda, *Algorithm 856: APPSPACK 4.0: Asynchronous parallel pattern search for derivative-free optimization*, ACM Transactions on Mathematical Software, 32 (2006), pp. 485–507.

[2] J. D. Griffin and T. G. Kolda, *Nonlinearly-constrained optimization using asynchronous parallel generating set search*, Tech. Rep. SAND2007-3257, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, May 2007. To appear in Applied Mathematics Research eXpress.

[3] J. D. Griffin, T. G. Kolda, and R. M. Lewis, *Asynchronous parallel generating set search for linearly-constrained optimization*, SIAM Journal on Scientific Computing, 30 (2008), pp. 1892–1924.

[4] T. G. Kolda, *Revisiting asynchronous parallel pattern search for nonlinear optimization*, SIAM Journal on Optimization, 16 (2005), pp. 563–586.

[5] T. G. Kolda, R. M. Lewis, and V. Torczon, *Optimization by direct search: New perspectives on some classical and modern methods*, SIAM Review, 45 (2003), pp. 385–482.

[6] ——, *Stationarity results for generating set search for linearly constrained optimization*, SIAM Journal on Optimization, 17 (2006), pp. 943–968.


Sandia National Laboratories