SANDIA REPORT

SAND2004-6391
Unlimited Release
Printed December 2004

APPSPACK 4.0: Asynchronous Parallel Pattern Search for Derivative-Free Optimization

Genatha A. Gray and Tamara G. Kolda

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.doe.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/ordering.htm

SAND2004-6391
Unlimited Release
Printed December 2004

APPSPACK 4.0: Asynchronous Parallel Pattern Search for Derivative-Free Optimization

Genetha A. Gray and Tamara G. Kolda
Computational Sciences and Mathematics Research Department
Sandia National Laboratories
Livermore, CA 94551–9217
{gagray,tgkolda}@sandia.gov

Abstract

APPSPACK is software for solving unconstrained and bound constrained optimization problems. It implements an asynchronous parallel pattern search method that has been specifically designed for problems characterized by expensive function evaluations. Using APPSPACK to solve optimization problems has several advantages: No derivative information is needed; the procedure for evaluating the objective function can be executed via a separate program or script; the code can be run in serial or parallel, regardless of whether or not the function evaluation itself is parallel; and the software is freely available. We describe the underlying algorithm, data structures, and features of APPSPACK version 4.0 as well as how to use and customize the software.

Contents

Figures

APPSPACK 4.0: Asynchronous Parallel Pattern Search for Derivative-Free Optimization

1 Introduction

APPSPACK is software for solving unconstrained and bound-constrained optimization problems, i.e., problems of the form

$$\begin{array}{ll} \min & f(\mathbf{x}) \\ \text{subject to} & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}. \end{array} \tag{1}$$

Here, $f : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ and $\mathbf{x} \in \mathbb{R}^n$. The upper and lower bounds are optional on an element-by-element basis; specifically, $\mathbf{l}$ is an n-vector with entries in $\mathbb{R} \cup \{-\infty\}$ and $\mathbf{u}$ is an n-vector with entries in $\mathbb{R} \cup \{+\infty\}$. To find a solution of (1), APPSPACK implements asynchronous parallel pattern search (APPS) [Hough et al. 2001; Kolda 2004], a method in the class of direct search methods [Wright 1996; Lewis et al. 2000]. APPS is a variant on generating set search as described by Kolda et al. [2003]. Since it is a direct search method, APPS does not require any gradient information and is thus applicable to a variety of contexts. APPS is provably convergent if the underlying objective function is suitably smooth [Kolda and Torczon 2003; Kolda and Torczon 2004; Kolda 2004].

APPSPACK is targeted to simulation-based optimization. These problems are characterized by a relatively small number of variables (i.e., $n < 100$), and an objective function whose evaluation requires the results of a complex simulation. One standard application of this kind of optimization is parameter estimation. The goal in this case is to identify the set of simulator input parameters that produces output that most closely matches some given observed data. For this problem, the objective function might be of the form

$$f(\mathbf{x}) = \sum_{i=1}^{N} \left(s_i(\mathbf{x}) - o_i \right)^2 .$$

Here, N denotes the number of data points to be compared; for example, the points could correspond to times or spatial locations. The values o_i for $i = 1, \ldots, N$ are the given observed data values at these points, and the values $s_i(\mathbf{x})$ for $i = 1, \ldots, N$ are the simulator outputs at the same points, depending on the input $\mathbf{x}$. Note that in order to discover the $\mathbf{x}$ that yields the best fit to the observed data, multiple simulations are required.

Using APPSPACK to solve optimization problems has the following advantages:

- No derivative information in needed.
- The procedure for evaluating the objective function does not need to be encapsulated in a subroutine and can, in fact, be an entirely separate program.
- The code can be run in serial or in parallel, regardless of whether or not the objective function itself runs in parallel.
- The software is freely available under the terms of the L-GPL.

These advantages have prompted users to employ APPSPACK for a wide variety of applications. See, for example, Hough et al. [2001], Mathew et al. [2002], Chiesa et al. [2004],Kupinksi et al. [2003], Croue [2003], Gray et al. [2003], and Fowler et al. [2004].

APPSPACK version 4.0 is based on the specific algorithm described by Kolda [2004]. It is written in C++ and uses MPI [Gropp et al. 1996; Gropp and Lusk 1996] for parallelism. The underlying algorithm and its class structures are described in Section 2. Potential users of APPSPACK will be most interested in Section 3, where we explain how to obtain, compile, install, and use APPSPACK, and in Section 5 where we summarize some successful applications. In addition, customizations of APPSPACK are discussed in Section 4. Sections 4.1 and 4.2 describe how the software can be directly linked with the objective function and how different means of communication can be used for the function evaluation. An ad hoc means of handling of general linear and nonlinear constraints is also discussed (see Section 4.3). Implementation of theoretically more robust ways to handle these types of constraints is underway and will be communicated in a future publication and software release.

The notation in this paper is as follows. A boldface capital letter, e.g., $\mathbf{T}$, denotes a set of vectors. A script capital letter, e.g., $\mathcal{I}$, denotes a set of indices. A boldface lowercase letter, e.g., $\mathbf{x}$, denotes a vector, and its ith entry is denoted by x_i. Note that $\mathbf{d}_i$ represents the ith vector in a set of vectors and not the ith component in $\mathbf{d}$, which would instead be denoted d_i (no boldface). Greek letters, e.g., α, represent scalars.

2 APPS algorithm & implementation

The APPS algorithm includes numerous details that are essential for efficient and correct implementation but not for basic understanding of the method. Omitting these details, APPS can be simply described as follows.

1. Generate a set of trial points to be evaluated,

$$\mathbf{T} = \{\mathbf{x} + \Delta_i \mathbf{d}_i : i \in \mathcal{I}\}. \tag{2}$$

   Here, $\mathbf{x}$ is the *best point* known so far, $\mathbf{d}_i$ is the ith *search direction*, Δ_i is the corresponding *step length*, and $\mathcal{I}$ is the subset of search directions for which new trial points should be generated.

2. Send the set $\mathbf{T}$ to the *conveyor* for evaluation, and collect a set of evaluated points, $\mathbf{E}$, from the conveyor. (The conveyor is a mechanism for shuttling trial points through the process of being evaluated.)

3. Process the set $\mathbf{E}$ to see if any point in $\mathbf{E}$ is "better than" $\mathbf{x}$. In other words, check if there exists a $\mathbf{y} \in \mathbf{E}$ such that

$$\mathbf{y} \prec \mathbf{x}, \tag{3}$$

   where the notation "$\prec$" denotes the "better than" relationship which essentially means that the point $\mathbf{y}$ has a lower function value. (It is described in detail in Figure 2.) If $\mathbf{E}$ contains a point better than $\mathbf{x}$, then the iteration is successful; otherwise, it is unsuccessful.

4. If the iteration is successful, replace $\mathbf{x}$ with the new best point (from $\mathbf{E}$). Optionally, regenerate the set of search directions and delete any pending trial points in the conveyor.

5. If the iteration is unsuccessful, reduce certain step lengths as appropriate. In addition, check for convergence based on the step lengths.

A detailed procedural version of APPS is given in Figure 1; for a complete mathematical description and analysis, see Kolda [2004]. The remainder of this section describes the implementation details and specific C++ objects of APPSPACK version 4.0.

Initialization.
Choose initial guess $\mathbf{x} \in \mathbb{R}^n$ and tolerances $\Delta_{\min}$ and Δ_{tol}.
Let $\mathbf{D} = \{\mathbf{d}_1, \ldots, \mathbf{d}_p\}$ be a set of search directions that conform to the nearby boundary.
Set $\Delta_i = \Delta_{\text{init}}$ and $\tau_i = -1$ for $i = 1, \ldots, p$.

Iteration.

(1) **Generate Trial Points.**

Let $\mathcal{I} := \{i \,:\, \Delta_i \geq \Delta_{\text{tol}} \text{ and } \tau_i = -1\}$. For each $i \in \mathcal{I}$:

- Let $\tilde{\Delta}_i \leq \Delta_i$ be the length of the longest feasible step from $\mathbf{x}$ along direction $\mathbf{d}_i$.
- Create a new trial point $\mathbf{y} := \mathbf{x} + \tilde{\Delta}_i \mathbf{d}_i$ and an associated tag.
- Along with $\mathbf{y}$, save the following information.
  - $\textsc{Parent\_Tag}(\mathbf{y}) := \textsc{Tag}(\mathbf{x})$
  - $\textsc{Direction\_Index}(\mathbf{y}) := i$
  - $\textsc{Step}(\mathbf{y}) := \Delta_i$
- Set $\tau_i := \textsc{Tag}(\mathbf{y})$.
- Add the new trial point to the set $\mathbf{T}$.

(2) **Exchange Trial Points.**

Send the (possibly empty) set of new trial points $\mathbf{T}$ to the evaluation conveyor.

Collect a set $\mathbf{E}$ of trial points that have been evaluated.

(3) **Process Evaluated Trial Points.**

Let $\mathbf{z}$ denote the "best" point in $\mathbf{E}$.

If $\mathbf{z}$ is "better than" $\mathbf{x}$, goto Step 4; otherwise, goto Step 5.

(4) **Successful Step.**

- Delete $\mathbf{x}$, remove $\mathbf{z}$ from $\mathbf{E}$, and reset $\mathbf{x} := \mathbf{z}$. Delete remaining points in $\mathbf{E}$.
- *Check for convergence based on the function value.*
- Compute search directions $\mathbf{D} = \{\mathbf{d}_1, \ldots, \mathbf{d}_p\}$ that conform to the nearby boundary (p may also change).
- Set $\Delta_i := \max\{\textsc{Step}(\mathbf{z}), \Delta_{\min}\}$ and $\tau_i := -1$ for $i = 1, \ldots, p$.
- Prune the evaluation queue.
- Go to Step 1.

(5) **Unsuccessful Step.**

- For each $\mathbf{y} \in \mathbf{E}$: If $\textsc{Parent\_Tag}(\mathbf{y}) = \textsc{Tag}(\mathbf{x})$, then let $i = \textsc{Direction\_Index}(\mathbf{y})$ and set $\Delta_i := \frac{1}{2}\Delta_i$ and $\tau_i := -1$.
- Delete all points in $\mathbf{E}$.
- *Check for convergence based on the step lengths.*
- Go to Step 1.

Figure 1. APPS Algorithm

2.1 Points

Points (i.e., the best point and trial points) are stored as `APPSPACK::Point` objects. As previously described, new points are generated according to equation (2), and each new point $\mathbf{y}$ is of the form

$$\mathbf{y} = \mathbf{x} + \Delta_i \mathbf{d}_i, \tag{4}$$

where $\mathbf{x}$ is the *parent*, Δ_i is the step length, and $\mathbf{d}_i$ is the direction. Besides the vector $\mathbf{y} \in \mathbb{R}^n$ itself, each `Point` object stores some additional relevant information. Every `Point` includes a unique *tag*, a positive integer that is used as an identifier. In addition, each `Point` contains information about the parent (i.e., the vector $\mathbf{x}$), search direction, and step length used to generate $\mathbf{y}$ according to equation (4). Once a trial point has been evaluated, its function value, $f(\mathbf{y}) \in \mathbb{R} \cup \{+\infty\}$, is also stored in `Point`. This function value is stored as a `APPSPACK::Value` object which was created to handle the possibility that $f(\mathbf{y}) = +\infty$. APPSPACK uses the special case $f(\mathbf{y}) = +\infty$ to signify trial points that could not be evaluated (e.g., the simulator failed) and certain types of infeasible points.

In order for a trial point $\mathbf{y}$ to be the next best point, it must satisfy two conditions. First, it must satisfy a *decrease condition* with respect to its parent, and second, it must have a function value that improves upon that of the current best point. We describe both of these conditions below.

A trial point $\mathbf{y}$ satisfies a decrease condition with respect to its parent $\mathbf{x}$ and step length Δ if the following holds:

$$f(\mathbf{y}) < f(\mathbf{x}) - \alpha\Delta^2, \tag{5}$$

where $\alpha \geq 0$ is a user-defined constant. If $\alpha = 0$, this is called *simple decrease* [Torczon 1995]; otherwise, if $\alpha > 0$, this is called *sufficient decrease* [Yu 1979; Lucidi and Sciandrone 2002]. To indicate whether or not a trial point satisfies (5) with respect to its parent, state information is stored in object `Point`. The state also specifies whether or not a trial point has been evaluated.

Satisfying decrease condition (5) is only part of the comparison APPSPACK uses to determine whether or not one point is "better than" another. The comparison also considers the corresponding function values and defines a scheme for tie-breaking in the case that these function values are equal. The complete procedure for determining if a point $\mathbf{y}$ is better than a point $\mathbf{x}$ (i.e. whether $\mathbf{y} \prec \mathbf{x}$) is detailed in Figure 2.

Let $\mathbf{y}$ and $\mathbf{x}$ be `Point` objects. The following procedure determines if $\mathbf{y} \prec \mathbf{x}$.

- If $\textsc{Tag}(\mathbf{y}) = \textsc{Tag}(\mathbf{x})$ (i.e., they are the same point), then $\mathbf{y} \not\prec \mathbf{x}$. Else, continue.
- If $\mathbf{y}$ does not satisfy (5) with respect to its parent, then $\mathbf{y} \not\prec \mathbf{x}$. Else, continue.
- If $\mathbf{x}$ does not satisfy (5) with respect to its parent, then $\mathbf{y} \prec \mathbf{x}$. (Note that $\mathbf{y}$ has been evaluated and does satisfy (5) by the previous bullet.) Else, continue.
- If $f(\mathbf{y}) < f(\mathbf{x})$, then $\mathbf{y} \prec \mathbf{x}$. Else, continue.
- If $f(\mathbf{y}) > f(\mathbf{x})$, then $\mathbf{y} \not\prec \mathbf{x}$. Else, continue.
- It must be the case that $f(\mathbf{y}) = f(\mathbf{x})$, so we break the tie by choosing the point with the lower tag. If $\textsc{Tag}(\mathbf{y}) < \textsc{Tag}(\mathbf{x})$, then $\mathbf{y} \prec \mathbf{x}$. Else $\mathbf{y} \not\prec \mathbf{x}$.

Figure 2. Determining if point $\mathbf{y}$ is "better" than point $\mathbf{x}$ ($\mathbf{y} \prec \mathbf{x}$).

In summary, an APPSPACK `Point` object stores the following information:

- the vector $\mathbf{y} \in \mathbb{R}^n$;
- its unique tag, denoted TAG($\mathbf{y}$);
- its parent's tag, denoted PARENT_TAG($\mathbf{y}$);
- its parent's function value;
- the step used to produce it, denoted STEP($\mathbf{y}$);
- the index of the direction used to produce it, denoted DIRECTION_INDEX($\mathbf{y}$).
- the state information;
- its function value $f(\mathbf{y})$ (if it has been evaluated).

The state information indicates whether or not the point has been evaluated and, if it has, whether or not it satisfies the sufficient decrease condition. Note that the parent vector is not explicitly stored; instead, only its corresponding tag and function value are stored. Likewise, the actual direction vector $\mathbf{d}_i$ is not stored; instead, only its index i is stored.

2.2 Constraints and Scaling

Various types of constraints can potentially be handled by APPSPACK using the abstract interface defined by the `APPSPACK::Constraints::Interface` class. Currently, we support only bound constraints via the `APPSPACK::Constraints::Bounds` class, but have plans to support general linear and nonlinear constraints using alternative constraint classes in the future. The constraints object is passed as an argument to the `APPSPACK::Solver`. We employ this programming structure to leave open the possibility of providing user-developed constraint classes in future versions.

The bounds on the variables are specified by the user in the APPSPACK input file (see Section 3.3). They are used both to determine a conforming set of search directions (see Section 2.3) and to generate trial points (see Section 2.4).

Related to the bounds and stored in the same object is the variable scaling. Because derivative-free methods do not use any gradient information, proper scaling of the variables is critical. Although scaling is not explicitly mentioned in the description of APPS provided in Figure 1, it plays important roles in convergence (see Section 2.5), determining a conforming set of search directions (see Section 2.3), calculating trial points (see Section 2.4), and looking up points in the cache (see Section 2.9).

To define a default scaling vector, APPSPACK uses the bounds. Specifically, let $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ denote the vectors of the lower and upper bounds, respectively. Then, the components of the scaling vector $\mathbf{s} \in \mathbb{R}^n$ are defined as

$$s_i = u_i - l_i \text{ for } i = 1, \ldots, n. \tag{6}$$

In the case that all the bounds are finite, the user may choose either to use this default scaling vector or to provide one in the APPSPACK input file ("Scaling" in the "Bounds" sublist). If finite bounds are not provided or do not exist, $\mathbf{s}$ must be provided by the user. Note that this approach to scaling was motivated by another derivative-free optimization software package, IFFCO Choi et al. [1999], a Fortran implementation of the implicit filtering method Gilmore and Kelley [1995; Kelley [1999].

2.3 Search Directions and Step Lengths

The search directions are handled by the `APPSPACK::Directions` class. A new set of search directions, or a search pattern, is computed every time an iteration is successful (see Step 4 in Figure 1). To generate this search pattern, APPSPACK considers the scaled coordinate directions and excludes directions outside the tangent cone. Specifically, the set of search directions is defined as

$$\mathbf{D} = \{\mathbf{d}_1, \ldots, \mathbf{d}_p\} = \{s_i \mathbf{e}_i : x_i < u_i\} \cup \{-s_i \mathbf{e}_i : x_i > \ell_i\},$$

where $\mathbf{d}_i$ is the ith search direction and $\mathbf{e}_i$ is the ith unit vector.

Each direction $\mathbf{d}_i$ has a tag, τ_i, and step length, Δ_i, associated with it. The tag is an integer that indicates whether or not there are any points in the evaluation conveyor associated with a given search direction. If $\tau_i = -1$, then there are currently no unevaluated trial points with parent $\mathbf{x}$ that were generated using direction $\mathbf{d}_i$. In this case, the step length Δ_i is the value that will be used in Step 1 of Figure 1 to compute a new trial point in direction i. Otherwise, τ_i is the tag number of the point associated with direction $\mathbf{d}_i$, and Δ_i is the step length that was used to generate that point.

2.4 Generation of Trial Points

Trial points are generated in the `APPSPACK::Solver` class. As indicated in Step 1 of Figure 1, a trial point is computed for each direction $i \in \mathcal{I}$ where $\mathcal{I} = \{i \ : \ \Delta_i \geq \Delta_{\text{tol}} \text{ and } \tau_i = -1\}$. In other words, the set $\mathcal{I}$ contains the search directions that have not yet converged (see Section 2.5) and do not currently have a trial point in the evaluation conveyor.

For each $i \in \mathcal{I}$, a feasible trial point is calculated. If $\mathbf{y} = \mathbf{x} + \Delta_i \mathbf{d}_i$ is not feasible, then an appropriate *pseudo-step* must be determined. The pseudo step, $\tilde{\Delta}_i$, is the longest possible step that is feasible, and it is formally defined as

$$\tilde{\Delta}_i = \max\{\Delta \in [0, \Delta_i] \ : \ \mathbf{l} \leq \mathbf{x} + \Delta \mathbf{d}_i \leq \mathbf{u}\}.$$

2.5 Stopping Conditions

The primary stopping condition in the APPS algorithm is based on the step length. This criteria was chosen because it can be shown that if the objective function is suitably smooth, then the gradient can be bounded as multiple of the step size, i.e., $\|\nabla f(\mathbf{x})\| = O(\max_i\{\Delta_i\})$ [Kolda et al. 2003]. In other words, the steps only get smaller if the norm of the gradient is decreasing. Hence, the step length can be used to define a stopping requirement.

The stopping condition based on step length is used in Step 5 of Figure 1. Specifically, APPS converges if *all* the step lengths are less than the specified tolerance; i.e.,

$$\Delta_i < \Delta_{\text{tol}} \text{ for } i = 1, \ldots, p. \tag{7}$$

Here, we say that the ith direction is converged if $\Delta_i < \Delta_{\text{tol}}$. The tolerance Δ_{tol} can be specified by the user in the APPSPACK input file ("Step Tolerance" in the "Solver" sublist). The default value is $\Delta_{\text{tol}} = 0.01$ which corresponds to a 1% change in the variables when the default scaling defined in (6) is used.

An alternative stopping condition is based on whether or not the function has reached a specified threshold. This criteria may be useful when the desired minimum value is known. For example, in the parameter estimation problem described in Section 1, it may be reasonable to stop when $f(\mathbf{x}) < 0.03$ or when the fit is within 3% of being exact. Step 4 of Figure 1 shows the implementation of the

function tolerance stopping criteria. Specifically, the iterations are terminated if

$$f(\mathbf{x}) \leq f_{\text{tol}}, \tag{8}$$

where f_{tol} is defined by the user in the APPSPACK input file ("Function Tolerance" in the "Solver" sublist). By default, this stopping condition is not employed.

Stopping can also be defined in terms of the number of function evaluations. In other words, the algorithm can be discontinued after a specified number of function evaluations has been completed. This sort of stopping criteria might be useful when the function evaluations are based on a particularly expensive simulation, and the user wants to adhere to a specified budget of evaluations. By default, this stopping criteria is not used, but it can be activated by the user by specifying a maximum number of function evaluations in the APPSPACK input file ("Maximum Evaluations" in the "Solver" sublist).

2.6 The Evaluation Conveyor

From the point of view of the APPS algorithm, the evaluation conveyor simply works as follows: A set of unevaluated points **T** is exchanged for a set of evaluated points **E** (Step 2 of Figure 1). The input set of unevaluated points may be empty. However, because returning an empty set of evaluated points means that the current iteration cannot proceed, the output set of evaluated points must always be non-empty.

Within the conveyor, a trial points moves through three stages. The first stage is to wait in a holding pen (the "Wait" queue) until resources become available for evaluating its function value. The second stage occurs while the function value is in the process of being computed, (in which case it sits in the "Pending" queue). The third stage takes place after the evaluation has been completed, while the trial point waits to be returned as output from the conveyor (in the "Return" queue). Each of these stages is described in more detail below.

One key point in this process is that it may take more than one iteration for a point to move through the conveyor. Thus, the set of points **T** that is input is not necessarily the same as the set of points **E** that is output. Furthermore, because it may take multiple iterations for a point to move through the conveyor, it is often desirable to remove some or all of the points that are in the first stage of the conveyor, waiting for evaluation. This removal of points is called pruning.

Every point that is submitted to the evaluation conveyor is eventually either returned or pruned. Pruning occurs in Step 4 of Figure 1 while returning takes place in Step 2. The evaluation conveyor facilitates the movement of points through the following three queues:

- **W**, the "Wait" queue where trial points wait to be evaluated. This is the only queue from which points can be pruned.
- **P**, the "Pending" queue for points with on-going evaluations. Its size is restricted by the resources available for function evaluations.
- **R**, the "Return" queue where evaluated points are collected. Its size can be controlled by the user.

This conveyor process is handled by the `APPSPACK::Conveyor` object. Each time a set of trial points is received (Step 2 of Figure 1), the conveyor follows the procedure diagrammed in Figure 3.

Points that are waiting to be submitted to the executor for evaluation are stored in **W**, and they remain there until either there is space in **P** or they are pruned. The **W** queue is pruned whenever an iteration is successful (Step 4 of Figure 1). By default, pruning is defined as the emptying of **W**.

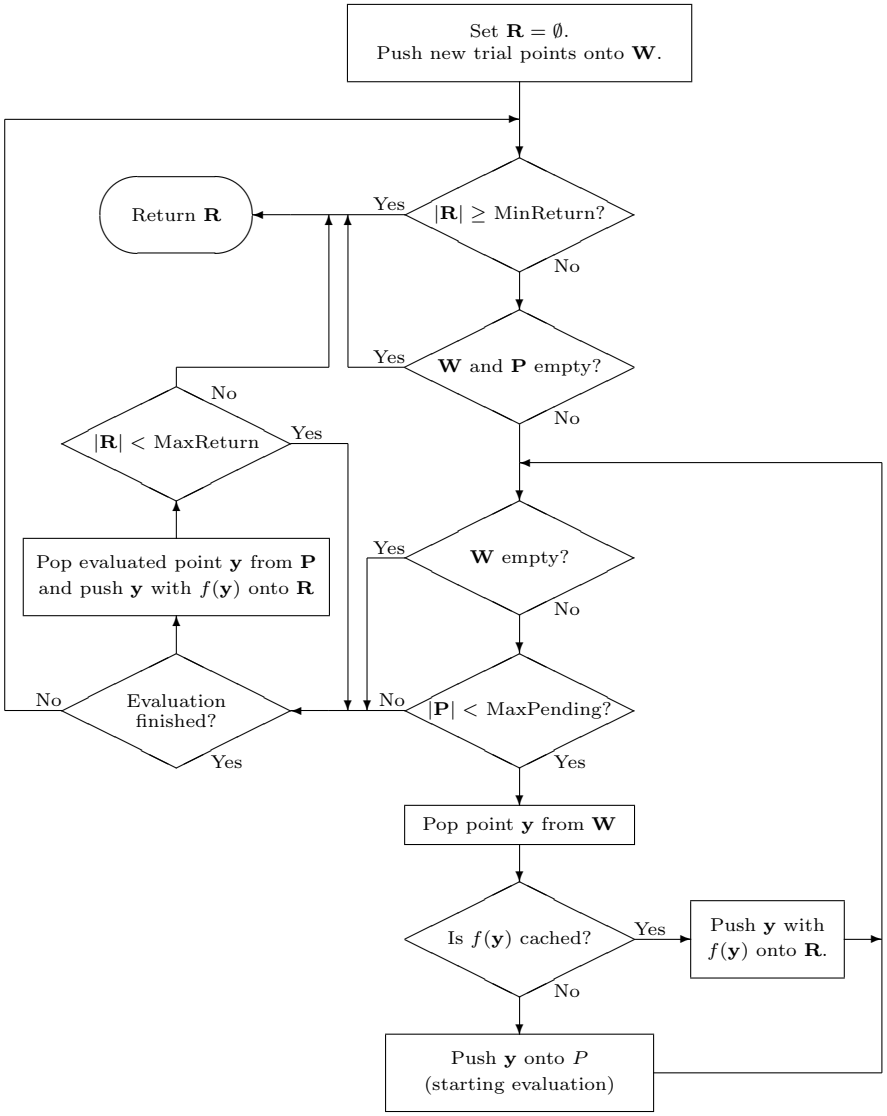

Figure 3. Conveyor actions for trial point exchange

However, the results of a pruning can be modified by setting the "Max Queue Size" parameter in the "Solver" sublist of the APPSPACK input file. Here, the user can specify the number of points that should remain in **W** after it is pruned. In this case, the oldest points are deleted and the newest points remain in the queue.

Before a point moves from **W** to **P**, the cache is checked to see if the function value has already been calculated for that point (see Section 2.9). If so, the cached function value is obtained, and the point is moved directly to **R**. If not, the point moves to **P** and is evaluated. Once a point has been pushed from **W** onto **P** or **R**, it can no longer be pruned.

Points which have been submitted to the executor for evaluation are stored in **P**. The executor handles the distribution of evaluation tasks to workers and is described in Section 2.7. The size of **P** is solely determined by the executor and depends on available resources. Recall that APPSPACK was designed to accommodate problems that may have expensive function evaluations. Hence, points may remain in **P** for several iterations. Once the executor returns the results of the function evaluation, the point is moved to **R**.

Essentially, the conveyor process continues until *enough* evaluated points are collected in the **R**. Enough is defined by the "Minimum Exchange Return" value set in the "Solver" sublist of the APPSPACK input file. The default value is one, but larger values can be used to force the conveyor to collect more evaluated trial points before returning. In the extreme, the conveyor process can continue until every trial point has been evaluated and collected. This behavior defines a synchronous pattern search [Lewis and Torczon 1996] and can be activated by setting the parameter "Synchronous" to true in the "Solver" sublist. (The default is false.) Finally, note that the size of **R** can also be controlled by defining a "Maximum Exchange Return" size in the "Solver" sublist, which defaults to 1000.

2.7 Executors

When a point enters the second stage of the conveyor, the **P** queue, it must be assigned to a worker (if running in parallel) and evaluated. The executor coordinates the assignment of points to workers for function evaluation. Its pure virtual abstract interface is defined in `APPSPACK::Executor::Interface` and includes the following:

- A boolean function that returns true if the executor is waiting to spawn more function evaluations. The result of this function is used in the conveyor test $|\mathbf{P}| <$ MaxPending shown in Figure 3.
- A spawn function that initiates a function evaluation. The input is a vector $\mathbf{y}$ and its corresponding tag, $\textsc{Tag}(\mathbf{y})$. The tag is used as an input argument in the function evaluation executable, and it is needed to match the resulting function value with the appropriate point in queue **P**.
- A receive function that checks whether or not any function evaluations have finished. If an evaluation has been completed, the output is the objective function value and any related information.

The executor is passed as an argument to the `APPSPACK::Solver` and may be customized by the user. APPSPACK 4.0 contains two concrete implementations of the executor, `APPSPACK::Executor::Serial` and `APPSPACK::Executor::MPI`.

The serial executor does exactly as its name implies— executes function evaluations one at a time. In other words, the size of the pending queue **P** is exactly one and all evaluations are performed immediately on the current processor (because there are no workers). The spawn operation calls

the evaluator (described in Section 2.8) directly. An example of an APPSPACK executable for serial mode is shown in Figure 4. While there may be some situations where the serial version of APPSPACK is useful, this mode is provided primarily for testing purposes.

```
// Construct empty parameter list
APPSPACK::Parameter::List params;

// Parse input file to fill parameter list
// (argv[1] contains the name of the input file)
APPSPACK::parseTextInputFile(argv[1], params);

// Construct evaluator object using parameters from the input file
APPSPACK::Evaluator::SystemCall evaluator(params.sublist("Evaluator"));

// Construct serial executor using the just-constructed evaluator
APPSPACK::Executor::Serial executor(evaluator);

// Construct bounds object using parameters from the input file
APPSPACK::Constraints::Bounds bounds(params.sublist("Bounds"));

// Construct solver object using parameters from the input file and
// the just created executor and bounds.
APPSPACK::Solver solver(params.sublist("Solver"), executor, bounds);

// Solver the optimization problem
solver.solve();

// Access the answer
vector<double> x = solver.getBestX();
bool isF = solver.isBestF();
double f = solver.getBestF();
```

Figure 4. Calling APPSPACK in Serial Mode

In contrast, the MPI executor spawns function evaluations to workers that execute as separate processes. In this version, the worker processes must be started independently in the main program. Figure 5 contains a documented example of a default executable for the MPI manager, and Figure 6 gives a corresponding MPI worker executable. Both of these executables can be customized by the user. The parallel version of APPSPACK is preferred and is how it is intended to be used.

2.8 Evaluators

The actual objective function evaluation of the trial points is handled by the abstract interface defined in the `APPSPACK::Evaluator::Interface` class. This structure allows the user to either use the provided default evaluator or to create a customized one. The default evaluator can be found in the `APPSPACK::Evaluator::SystemCall` class, and it works as follows: A function input file containing the point to be evaluated is created. Then, an external system call is made to the user-provided executable that calculates the function value. After the function value has been computed, the evaluator reads the result from the function output file. Finally, both the function input and output files are deleted. This process is illustrated in Figure 7. Information regarding the user

```
// Construct empty parameter list
APPSPACK::Parameter::List params;

// Parse input file to fill parameter list
APPSPACK::parseTextInputFile(argv[1], params);

// Send the evaluator information to each worker
APPSPACK::GCI::initSend();
params.sublist("Evaluator").pack();
for (int i = 0; i < nWorkers; i ++)
  APPSPACK::GCI::send(APPSPACK::Executor::MPI::Init, i+1);

// Construct MPI executor.
// The evaluator will be constructed on the workers and is not passed
// as an argument to the executor.
APPSPACK::Executor::MPI executor;

// Construct bounds object using parameters from the input file
APPSPACK::Constraints::Bounds bounds(params.sublist("Bounds"));

// Construct solver object using parameters from the input file and
// the just created executor and bounds.
APPSPACK::Solver solver(params.sublist("Solver"), executor, bounds);

// Solver the optimization problem
solver.solve();

// Send a termination command to each worker
APPSPACK::GCI::initSend();
for (int i = 0; i < nWorkers; i ++)
  APPSPACK::GCI::send(APPSPACK::Executor::MPI::Terminate, i+1);

// Access the answer
vector<double> x = solver.getBestX();
bool isF = solver.isBestF();
double f = solver.getBestF();
```

Figure 5. Calling APPSPACK in MPI Mode (Manager)

```
// Unpack information sent by manager
APPSPACK::Parameter::List params;
APPSPACK::GCI::recv(APPSPACK::Executor::MPI::Init, 0);
params.unpack();

// Construct evaluator using parameters from message
APPSPACK::Evaluator::SystemCall evaluator(params);

// Continuously receive and process incoming messages
while (1)
{
  // Blocking receive for the next message
  int msgtag, taskid;
  APPSPACK::GCI::recv();
  APPSPACK::GCI::bufinfo(msgtag, taskid);

  // Check for termination
  if (msgtag == APPSPACK::Executor::MPI::Terminate)
    break;

  // Local vars to be packed and unpacked
  int tag;
  vector<double> x;
  bool isF;
  double f;
  string msg;

  // Unpack the latest message
  APPSPACK::GCI::unpack(tag);
  APPSPACK::GCI::unpack(x);

  // Evaluate the function
  evaluator(tag,x,isF,f,msg);

  // Send a reply
  APPSPACK::GCI::initSend();
  APPSPACK::GCI::pack(tag);
  APPSPACK::GCI::pack(isF);
  APPSPACK::GCI::pack(f);
  APPSPACK::GCI::pack(msg);
  APPSPACK::GCI::send(APPSPACK::Executor::MPI::Feval,0);
}
```

Figure 6. APPSPACK MPI Worker

provided executable and the formats of the function input and output files is given in Section 3.2.

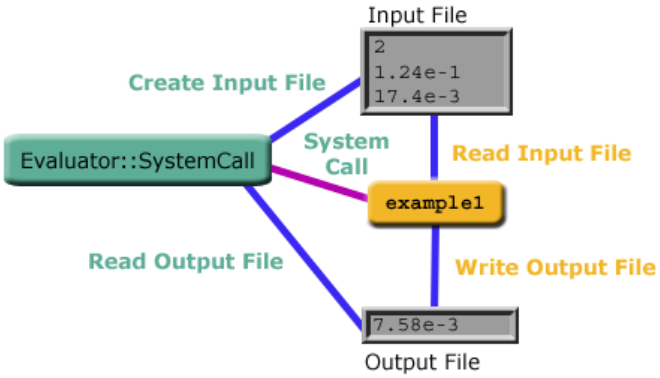

Figure 7. The "system call" evaluator.

The evaluator is its own entity and is not part of the executor nor is it directly called by the executor. In MPI mode, it is the job of the executor to assign a point to a worker for evaluation. Then, it is actually the worker that calls the evaluator, not the executor. Therefore, it is the job of the manager process to relay any necessary information about the function evaluation to the workers before the APPSPACK::Solver::solve function is called. For example, the evaluator informs the workers of the name of the executable to be used for the function evaluation. In serial mode, the executor has its own evaluator which is called in the spawn function.

By default version, APPSPACK assumes that the function evaluations should run as separate executables and that communication with the evaluation executable be done via file input and output. In other words, each worker makes an external system call as illustrated in Figure 8. This default design ensures applicability of APPSPACK to simulation-based optimization. Required simulations are often too complex to be easily or reasonably encapsulated into a subroutine. In this case, the capability of external system calls is essential. Moreover, allowing the user to supply a separate executable improves the usability of APPSPACK. For example, the user can write the program for the function evaluation in any language or can simply provide a script that executes individual steps of a complicated function evaluation. Finally, although system calls and file I/O do add to the overall run time, the computational time needed to complete a simulation is often such that the added time of external system calls and file I/O is negligible.

Despite the advantages of using system calls and file I/O, there are some applications for which the user may prefer to eliminate this overhead. For example, the function evaluation may be relatively inexpensive. To eliminate the external system calls, the user can provide a customized evaluator as detailed in Section 4.1.

2.9 Cache

Because the APPS algorithm is based on searching a pattern of points that lie on a regular grid, the same point may be revisited several times. Thus, to avoid evaluating the objective function at any point more than once, APPSPACK employs a function value cache. Each time a function evaluation is completed and a trial point is placed in the return queue **R**, the conveyor sends this point and its corresponding function value to the cache. Then, before sending any point to the pending queue **P**, the conveyor first checks the cache to see if a value has already been calculated. If it has, the cached function value is used instead of repeating the function evaluation.

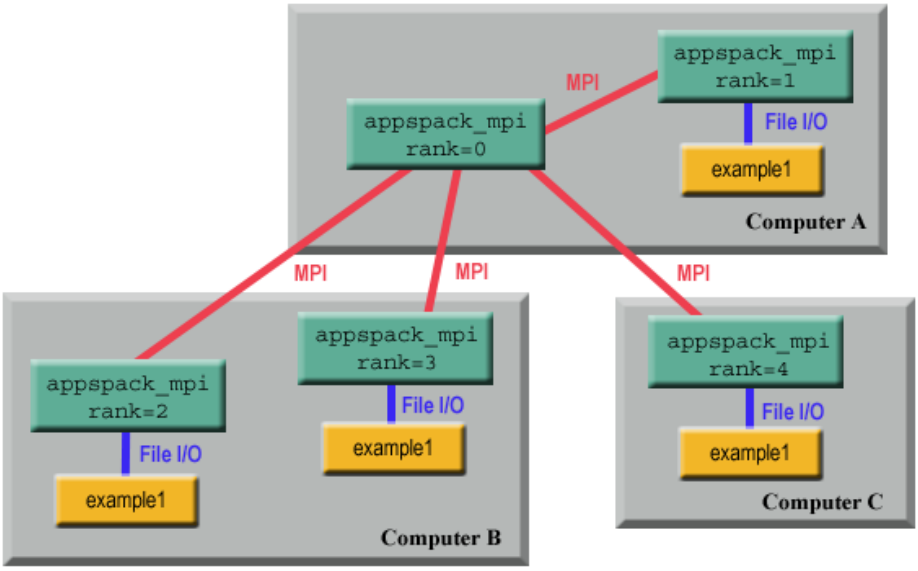

Figure 8. Parallel APPSPACK using the "system call" evaluator.

The cache operations are controlled by the `APPSPACK::Cache::Manager` class. Its functions include inserting new points into the cache, performing lookups, and returning previously calculated function values. Optionally, the cache manager can also create an output file with the contents of the cache or read an input file generated by a previous run. These features can be activated using the "Cache Output File" and the "Cache Input File" parameters of the "Solver" sublist in the APPSPACK input file.

Like its predecessors, APPSPACK version 4.0 uses a splay tree to store points in the cache [Hough et al. 2000]. A splay tree is a binary search tree that uses a series of rotations to move any accessed node to the root (see [Sleator and Tajan 1985]). Because the most recently accessed nodes are kept near the root of the tree, searching these nodes is fast. The APPS algorithm can take advantage of this characteristic of splay trees because it normally only revisits points that were evaluated recently.

Cache points are stored as `APPSPACK::Cache::Point` objects which include only the vector itself and its corresponding function value. This structure eases storage and comparison. The `Cache::Point` class includes some rules for point comparison rules which are explained below.

In order for two points to be deemed "equal," they must satisfy a cache comparison test. Our definition of equal allows for the possibility that all points are not represented with the same precision or that the function evaluations may include some approximations. Specifically, trial points $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, are "equal" if

$$|x_i - y_i| <= \varepsilon * s_i \text{ for } i = 1, \ldots, n$$

where s_i is the ith component of the scaling vector $\mathbf{s}$ and ε is some tolerance. The value of ε can be set by the user using the "Cache Comparison Tolerance" parameter in the "Solver" sublist of the APPSPACK input file, and defaults to half the stopping tolerance Δ_{tol}.

It is also necessary to impose some order on the points so that they can be placed in the tree. This cache comparison test is also based on the tolerance ε and the scaling vector $\mathbf{s}$. Here, trial

point $\mathbf{x}$ is less than $\mathbf{y}$ if there exists an index j such that

$$|x_i - y_i| < \varepsilon * s_i \text{ for } i = 1, \ldots, j-1 \text{ and } y_j - x_j > \varepsilon * s_i.$$

In other words, for ordering purposes, $\mathbf{x}$ comes before $\mathbf{y}$ if the first $j-1$ coordinates of $\mathbf{x}$ and $\mathbf{y}$ are equal (by our definition), and the jth coordinate of $\mathbf{y}$ is sufficiently greater than the jth coordinate of $\mathbf{x}$.

3 Using APPSPACK

3.1 Download, Compiling, and Installing APPSPACK

APPSPACK 4.0 is available as a free download under the terms of the GNU Lesser General Public License [1]. To download APPSPACK 4.0, follow these instructions:

- Go to `http://software.sandia.gov/appspack/version4.0/`.
- Click on "APPSPACK License, Version, and Download Information."
- Click on "APPSPACK 4.0."
- When prompted, save the file `appspack-4.0.tar.gz` to disk.

To unpack the compressed tar file, type

```
gunzip -c appspack-4.0.tar.gz | tar -xvf -
```

This will create a directory named `appspack-4.0` containing the source code for APPSPACK. Rename this directory `appspack`. The full path of the resulting `appspack` directory will be referred to as `${APPSPACK}` for the remainder of this paper.

The next step is configuring APPSPACK for your computing environment. To do this, go to the `${APPSPACK}` directory and type

```
./configure [options]
```

A full list of configure options can be obtained by typing `configure --help` or by consulting the online documentation. The most important option is

```
--with-mpi-compilers[=PATH]
```

which specifies that the MPI compilers (`mpicc`, `mpif77`, and `mpicxx` or `mpiCC`) should be used. The optional variable `PATH` designates the location of these compilers. Specifying the path is useful when multiple versions of MPI are available. If no path is indicated, the compilers found in the default path are used. If APPSPACK is configured without any MPI options, only the serial version will be compiled and installed.

Once APPSPACK has been successfully configured, the libraries and executables can be built. To compile APPSPACK, go to the `${APPSPACK}` directory, and type

[1] `http://www.gnu.org/copyleft/lesser.html`

```
make
```

Assuming that an MPI configure option was specified, the make command will create both the serial and MPI executables:

```
${APPSPACK}/src/appspack_serial
${APPSPACK}/src/appspack_mpi
```

The APPSPACK library `${APPSPACK}/srclibappspack.a` will also be created, and several examples will be compiled in the `${APPSPACK}/examples` directory.

Optionally, APPSPACK can be installed via the `make install` command. The default installation location is `/usr/local`, but this can be changed using the `--prefix` configure option. The executables `appspack_serial` and `appspack_mpi` are installed in the `bin` subdirectory, the header files are installed in the `include` subdirectory, and the library is installed in the `lib` directory. The examples are not installed.

3.2 Creating a Function Evaluation for APPSPACK

An executable for evaluating the objective function must be provided by the user. It can be a single program or a script that, in turn, calls other programs. APPSPACK calls this executable repeatedly to evaluate different trial points via the C `system()` command.

The executable command line should accept three input arguments: an input file name, an output file name, and a tag. In other words, the the calling sequence is

```
<Executable Name> <Input File> <Output File> <Tag>
```

The input file is created by APPSPACK to be read by the executable and is simply formatted. The first line is an integer that indicates the length of the vector, and each of the subsequent lines contain one component of the vector. For example, the APPSPACK input file containing the vector `[1.24e-1, 17.4e-3]` $\in \mathbb{R}^2$ is:

```
2
1.24e-1
17.4e-3
```

The output file is created by the executable and read by APPSPACK. It contains either the function value as a single numeric entry or an error string. An example output file containing a function value is:

```
7.58e-3
```

An example output file with an error string is:

```
Meshing Error
```

APPSPACK has the ability to tabulate different error strings and can track an unlimited number of messages. Strings are defined by the user in the executable. They can be more than one word but

must be only one line, and the string "Success" is disallowed as an error string. This feature may be useful for identifying the reasons for function evaluation failure.

The MPI version of APPSPACK executes multiple function evaluations in parallel. To prevent these parallel processes from overwriting each other's files, each call to the executable includes *uniquely named* input and output files. This is accomplished using the tag number associated with a point to name its related files. The tag is also provided as the third input argument of the executable to allow the user to uniquely name any additional files that may be created during the evaluation of the objective function. Note that while the tags and subsequent file names are unique for a single run of APPSPACK, they are not unique across repeated runs.

After a function evaluation has been completed, APPSPACK will automatically delete the input and output files. These files should *not* be deleted by the executable. However, any additional files that were created in the process of evaluating the function should be deleted by the executable. "Leftover" files, i.e., temporary files that are not deleted, are a frequent source of errors and may cause system disk space problems.

3.3 Creating the APPSPACK Input File

The user must provide an input file that specifies the parameters for running APPSPACK. There are three categories of input parameters: Evaluator, Bounds, and Solver. Each is described in the subsections that follow. To clarify our discussion of the general formatting of an APPSPACK input file, consider the example file shown here:

```
# SAMPLE APPSPACK INPUT FILE
@ "Evaluator"
"Executable Name" string "example1"
@@
@ "Bounds"
"Lower" vector 3 0 -0.6 0
"Upper" vector 3 5.7 0 1.3
"Is Lower" vector 3 0 1 1
"Is Upper" vector 3 1 0 1
"Scaling" vector 3 1.0 1.0 1.0
@@
@ "Solver"
"Initial X" vector 3 2e-1 0.3 0.5
"Synchronous" bool false
@@
```

Any empty lines are ignored, and comment lines begin with a `#`. The beginning of a category is specified by a `@` followed by a space and then the category name in quotes. The end of a category is designated by a line containing the symbols `@@`. Within a category, parameters are defined by name (in quotes), type (`string`, `int`, `double`, `vector`, or `bool`,), and value. String values, such as the executable name, are given in quotes. Double and integer values are just numbers. Vectors values include an integer that represents their length followed by each entry. Boolean values are written without quotes. The only recognized boolean values are `true` and `false`, and any other input is treated as false.

3.3.1 The `"Evaluator"` parameters

The three evaluator parameters are:

- `"Executable Name"` (string): Name of the executable that evaluates the objective function. This parameter should always be specified. If not, it defaults to `"a.out"`.
- `"Input Prefix"` (string): Input file prefix. The input files names are created by attaching a "." and a unique integer tag to this prefix. In general, this parameter does need to be specified, and it defaults to `"input"`.
- `"Output Prefix"` (string): Output file prefix. See `"Input Prefix."` Defaults to `"output"`.

3.3.2 The `"Bounds"` parameters

Since no gradient information is available to the APPS algorithm, it is critical that the variables be reasonably scaled. In APPSPACK, this is accomplished using a scaling vector, **s**. If finite upper and lower bounds are provided for each of the variables, then **s** is defined as

$$s_i = u_i - l_i, \text{ for } i = 1, \ldots, n.$$

where **u** and **l** denote the upper and lower bounds, respectively. If complete upper and lower bounds are not given, then the scaling must be provided by the user in the input file.

The bounds parameters are:

- `"Lower"` (vector): Vector of lower bounds.
- `"Upper"` (vector): Vector of upper bounds
- `"Scaling"` (vector): Scaling vector.
- `"Is Lower"` (vector): Binary (i.e., 0/1) vector. A zero in the i-th position indicates that there is no lower bound on the ith variable.
- `"Is Upper"` (vector): See `"Is Lower"`.

3.3.3 The `"Solver"` parameters

The user has the option of defining any number of the solver parameters. These parameters ensure the flexibility and applicability of APPSPACK to a wide variety of applications. Some of these parameters permit the user to customize the input or output of the algorithm and do not have any overall effect on the algorithm itself. Others allow the user to change some of the underlying mechanics of the method that may effect the algorithm's speed, but should not effect its convergence. Despite the number of choices offered, we haven't encountered any difficulties finding appropriate parameter values for any applications. In fact, the default values tend to work very well in most cases. Below, we review the solver parameters and their default values. More details can be found in the online documentation. We divide the solver parameters into several categories below for discussion.

The parameters called for in the initialization of the APPS algorithm outlined in Figure 1 can be defined using the following:

- `"Initial X"` (vector): The starting point. Default: a vector halfway between the upper and lower bounds.
- `"Initial F"` (double): Function value corresponding to the initial guess (if given). Note: It is *not* necessary to give an initial function value when specifying an `"Initial X"`.

- `"Step Tolerance"` (double): The stopping tolerance, Δ_{tol}. Default: 0.01.
- `"Minimum Step"` (double): The tolerance Δ_{min}. Default: twice the `"Step Tolerance."`
- `"Initial Step"` (double): The initial step length, Δ_{init}. Default: 1.0.
- `"Contraction Factor"` (double): The reduction factor in the step length after an unsuccessful function evaluation (used in Step 5 of Figure 1). Default: 0.5
- `"Bounds Tolerance"` (double): The tolerance used to determine if a bound constraint is active. Default: half the `"Step Tolerance"`.
- `"Sufficient Decrease Factor"` (double): Value of α in the sufficient decrease calculation, (5). Default: 0.01.

As discussed in Section 2.5, the primary stopping condition of APPSPACK is based on step length. However, two alternative stopping conditions are offered and can be activated with the following parameters:

- `"Function Tolerance"` (double): The function tolerance, f_{tol}, in (8). Default: Do not use this stopping criteria.
- `"Maximum Evaluations"` (int): Stop when the number of function evaluations exceeds this tolerance. Default: Do not use this stopping criteria.

The evaluation conveyor, described in Section 2.6, can be customized using the following parameters:

- `"Synchronous"` (bool): If set to true, APPS becomes *synchronous* parallel pattern search (PPS). Default: false.
- `"Max Queue Size"` (int): The number of points remaining in the pending queue $\mathbf{W}$ after a successful iteration and pruning. Default: 0.
- `"Minimum Exchange Return"` (int): Minimum number of values returned by the conveyor. Default: 1.
- `"Maximum Exchange Return"` (int): Maximum number of values returned by the conveyor. Default: max{ `"Minimum Exchange Return"`, 1000 }

There are also some options in how the cache is used. They are defined using the parameters:

- `"Cache Output File"` (string): Name of file for storing (cached) function values. This file can be used as input to future APPSPACK runs with the same objective function to prevent repeating evaluations. Default: no such file.
- `"Cache Input File"` (string): Name of file with cached function values, produced by a previous run of APPSPACK. The cache file must be produced by the same objective function. Default: no such file.
- `"Cache Comparison Tolerance"` (double): Value used to declare two points equal (with respect to the infinity-norm) in cache comparisons. Default: half the `"Step Tolerance"`.

Finally, the following parameters control the amount and format of the APPSPACK output.

- `"Debug"` (int): An integer specifying how verbose the output should be. The options range from 1 to 7, and higher values produce more output. Default: 3.
- `"Precision"` (int): Number of digits of precision in the output. Default: 3.

3.4 Running APPSPACK

Once an executable and an input file have been created, the user is ready to run APPSPACK. For the serial version, the command is:

```
${APPSPACK}/src/appspack_serial <input file>
```

For the MPI version, there is one master processor, and one or more workers to actually evaluate the objective function. The command line will vary a bit with the version of MPI. One example of a command line is

```
mpirun -np <integer> ${APPSPACK}/src/appspack_mpi <input file>
```

where `<integer>` is the number of processors that will be used.

Three examples are provided in the `${APPSPACK}/examples` directory. For all three examples, the objective function is

$$f(\mathbf{x}) = \sum_{i=1}^{n} \mu_i \, x_i^2.$$

In examples 1 and 2, $n = 2$ and $\mu_i = i$, and in example 3, $n = 3$ and $\mu_i = 1$. Example 1 includes only finite simple bounds on the variables while examples 2 and 3 include some nonlinear constraints. These will be discussed in more detail in Section 4.3. To run any of the examples, go to the examples directory, and type the appropriate command. For instance, the command line to run example 1 using the MPI version of APPSPACK on 3 processors might be

```
mpirun -np 3 ../src/appspack_mpi example1.apps
```

3.5 APPSPACK Output

To indicate the progress of the APPS algorithm, a wide range of data is available as output. Using the "Debug" parameter in the "Solver" sublist of the input file, the user can control what information is reported. The options range from 1 to 7 with the higher values producing more output. The default value is 3 which includes the initialization data, every new minimum, and the final solution. All seven debug levels are described in the online documentation. By default, all output is printed to standard output.

In the remainder of this section, we will describe the default APPSPACK output using an example. This output was produced running the MPI version of APPSPACK using 3 processors for example 1 from the `${APPSPACK}/examples` directory (as described in the previous section).

The APPSPACK output begins with a display of the list of input parameters that were used for the run. An example of this list is shown here:

```
##########################################
###   APPSPACK Initialization Results   ###

*** Parameter List ***
Bounds Tolerance = 0.005   [default]
Cache Comparison Tolerance = 0.005   [default]
Cache Input File = ""   [default]
Cache Output File = ""   [default]
```

```
Contraction Factor = 0.5   [default]
Debug = 3   [default]
Initial Step = 1   [default]
Initial X = [ 2.000e-01  3.000e-01 ]
Max Queue Size = 0   [default]
Maximum Exchange Return = 1000   [default]
Minimum Exchange Return = 1   [default]
Minimum Step = 0.02   [default]
Precision = 3   [default]
Step Tolerance = 0.01   [default]
Sufficient Decrease Factor = 0.01   [default]
Synchronous = false   [default]

*** Constraints ***

Bound Constraints
lower = [-1.000e+00 -1.000e+00 ]
upper = [ 1.000e+00  1.000e+00 ]
scaling = [ 2.000e+00  2.000e+00 ]

*** Conveyor ***

Using MPI Executor with 2 workers

### End APPSPACK Initialization Results ###
##########################################
```

The term `[default]` appears after each parameter for which no value was specified in the input file. The line under the `Conveyor` subheading indicates which version of APPSPACK was used (MPI or serial), and in the case of MPI, how many processors were employed as workers.

Subsequent output indicates the algorithm's progress. For example, our run produced the following:

```
New Min: f=<null>     x=[ 2.000e-01  3.000e-01 ] step=1.000e+00
 tag=0 state=Evaluated (Initial Point)

New Min: f= 1.180e+00 x=[ 1.000e+00  3.000e-01 ] step=1.000e+00
 tag=1 state=Evaluated  Success: 1

New Min: f= 1.800e-01 x=[ 0.000e+00  3.000e-01 ] step=5.000e-01
 tag=8 state=Evaluated  Success: 7

New Min: f= 8.000e-02 x=[ 0.000e+00 -2.000e-01 ] step=2.500e-01
 tag=19 state=Evaluated  Success: 13

New Min: f= 5.000e-03 x=[ 0.000e+00  5.000e-02 ] step=1.250e-01
 tag=28 state=Evaluated  Success: 19

New Min: f= 3.125e-04 x=[ 0.000e+00 -1.250e-02 ] step=3.125e-02
 tag=44 state=Evaluated  Success: 30
```

Each line indicates a new status for the algorithm. At the default debugging level, only new minimums (i.e. improvements on the best known point) are shown. At higher levels, the sets $\mathbf{T}$ of trial

points sent to conveyor or sets **E** of evaluated trial points are included in this list. Additionally, some information about the search directions can be included. Each of the output lines indicating the current status includes the function value, the point, its step length, tag, and state, and a message indicating how many iterations of each type have been completed. In this example, "Success" is the only message string used. In other examples, these counts may include error strings indicating infeasibility or simulation failure. (See Section 4.3 for an example.)

Finally, the output concludes with the results of the run and some summary information as shown here:

```
Final State: Step Converged

Final Min: f= 3.125e-04 x=[ 0.000e+00 -1.250e-02 ]
 step=3.125e-02 tag=44 state=Evaluated  Success: 30

Final Directions:
   0 : d = [ 2.000e+00  0.000e+00 ] step = 7.812e-03
   1 : d = [-2.000e+00  0.000e+00 ] step = 7.812e-03
   2 : d = [ 0.000e+00  2.000e+00 ] step = 7.812e-03
   3 : d = [ 0.000e+00 -2.000e+00 ] step = 7.812e-03

Number of Cached Function Evaluations: 9
Number of Evaluations: 37

Evaluation Breakdown by Message Type:
  Success: 37

Evaluation Breakdown by Processor and Message Type:
  Worker #1
    Success: 21
  Worker #2
    Success: 16
```

The `Final State` indicates why APPS algorithm terminated. In this case, it was because the stopping criteria based on step length was satisfied. Next, the final minimum is reported using the same status format of each new minimums, and the final search direction vectors and steps lengths are listed. Then, the total number of function evaluations completed is reported along with the number of times a saved function value was used instead of evaluating the function again. Lastly, the number of function evaluations is broken down by message type and processor.

3.6 User Resources

Several resources are available for users including online documentation, mailing lists, and a bug reporting system.

The online documentation for APPSPACK 4.0 is available at `http://software.sandia.gov/appspack/version4.0/`. It is created using doxygen[2], a tool that automatically generates documentation from the comments in the source code.

Users may be interested in subscribing to the APPSPACK-Announce or the APPSPACK-Users mailing list, or both. The announce list is a low-volume list for announcing new releases. The

[2] `http://www.doxygen.org/index.html`

users list is for general discussion. For more information, click on the "Mailing Lists" link at `http://software.sandia.gov/appspack/version4.0/`.

Bugs and enhancement requests are submitted using Bugzilla[3], a web-based defect tracking system created by The Mozilla Organization. The Bugzilla database for APPSPACK is located at `http://software.sandia.gov/bugzilla/`. To report a bug, click on "Enter a new bug report." (New users will need to register.) Next, choose APPSPACK from the list of products. Finally, input your bug or enhancement request. Descriptions should be as specific as possible.

4 Customizing APPSPACK

Since the serial version of APPSPACK is provided primarily for testing purposes, we focus our discussion of customizations on the MPI version.

4.1 Customizing the Function Evaluations

As described in Section 2.8, the default parallel version of APPSPACK requires that function evaluations be run as separate executables, and any communication between the workers and the executable must be done using file input and output.

Despite the advantages of using system calls and file I/O for simulation-based optimization, there are some applications for which the user may prefer to eliminate this overhead. For example, some function evaluations may require relatively large amounts of auxiliary data and the use of system calls would require that this data be re-read every evaluation. It may also be the case that that the function evaluation is cheap in comparison with the file I/O requirements.

For these applications, the user can create a customized evaluator class that derives from `APPSPACK::Evaluator::Interface` to be used in place of `APPSPACK::Evaluator::SystemCall` and directly compute the function value instead of making a call to an outside program. This also eliminates the need for function input and output files. An illustration of this customization is shown in Figure 9. Note that unlike the default evaluator shown in Figure 8, each worker actually executes the function evaluation itself.

Instructions for creating a custom evaluator are provided on the APPSPACK web site — see "Customizing APPSPACK" under "Related Pages."

4.2 Customizing the Parallelization

Customizing the evaluator only changes how individual functions are executed. In some cases, users may want to customize the way that the manager-worker relationship works. The key is in the `APPSPACK::Executor`. Two versions are provided with APPSPACK: `APPSPACK::Executor::Serial` and `APPSPACK::Executor::MPI`, for the serial and MPI versions of APPSPACK, respectively. A user can write a customized class that derives from `APPSPACK::Executor::Generic`. Note that customizing the executor may entirely eliminate the need for the evaluator.

Some motivations for customizing the executor are as follows.

- Multi-level parallelism can be achieved by assigning groups of processors to each function evaluation.

[3] `http://www.bugzilla.org/`

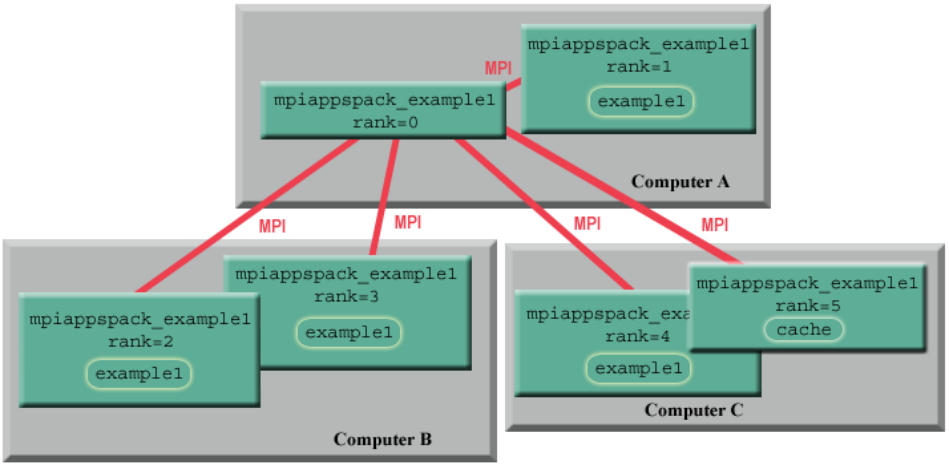

Figure 9. Parallel APPSPACK using a customized evaluator.

- The format of the MPI messages that are sent between the manager and the worker can be modified.
- The evaluation of functions can be handed off to another library such as a general optimization interface.

Instructions for creating a custom evaluator are provided on the APPSPACK web site — see "Customizing APPSPACK" under "Related Pages."

4.3 General Linear and Nonlinear Constraints

In APPSPACK 4.0, general linear and nonlinear constraints can be handled using the error string capabilities of the function output and a straight-forward modification of the function evaluation. Note that this customization only affects the function evaluation executable (or customized evaluator) and does not result in any changes to the APPS algorithm itself.

Let $\mathbf{y}$ be a trial point generated by the APPS algorithm. Then, to incorporate general linear or nonlinear constraints, use the following customized function evaluation procedure:

1. Check to see if $\mathbf{y}$ satisfies the constraint(s).
2. If $\mathbf{y}$ is feasible, evaluate the function.
3. If $\mathbf{y}$ is infeasible, write the appropriate error string.

Given the results of the function evaluation, either the function value or the error string, the APPS algorithm will continue as normal. In the case that the trial point is infeasible and an error message is returned, the function value will be handled as $f(\mathbf{y}) = +\infty$.

Both examples 2 and 3 in `${APPSPACK}/examples` contain bound constraints and general nonlinear constraints. The bound constraints are handled by the `APPSPACK::Constraints::Bounds` class as described in Section 2.2, and the nonlinear constraints are incorporated directly into the provided function evaluation routines. Consider the following line of output produced running example 2:

```
New Min: f= 1.005e+00 x=[-1.000e+00  5.000e-02 ] step=6.250e-02
tag=20 state=Evaluated  Constraint Violation: 3 Success: 11
```

Here, APPSPACK is tracking both the number of points that were not evaluated because they violated at least one of the constraints (`Constraint Violation`) and the number of feasible points that were successfully evaluated (`Success`). Moreover, the final function evaluation counts include a breakdown of feasible and infeasible points:

```
Number of Cached Function Evaluations: 7
Number of Evaluations: 26

Evaluation Breakdown by Message Type:
  Constraint Violation: 8
  Success: 18

Evaluation Breakdown by Processor and Message Type:
  Worker #1
    Constraint Violation: 5
    Success: 7
  Worker #2
    Constraint Violation: 3
    Success: 11
```

Using a customized function evaluation script to handle general linear and nonlinear constraints works well for the simple examples provided in APPSPACK. We also note that a similar customization of APPSPACK 4.0 was successful in solving the more complicated constrained optimization problems described by Fowler et al. [2004].

5 Conclusions

We have described the underlying algorithm, data structures, and features of APPSPACK version 4.0, a software package for solving unconstrained and bound-constrained optimization problems. Because APPSPACK does not require any derivative information, it is applicable to a wide variety of applications. Furthermore, since the procedure for evaluating the objective function can be an entirely separate, APPSPACK is well suited for simulation-based optimization or problems for which the evaluation of the objective function requires the results of a complicated simulation. Moreover, the software is freely available and can be easily downloaded, compiled and installed. The software has been written so that it is easily extensible and customizable to individual users' needs.

There are many real world applications that have demonstrated the benefits and suitability of APPSPACK. For example, Hough et al. [2001] applied APPSPACK to a thermal design problem for determining the settings for seven heaters in a thermal deposition furnace and to a 17 variable parameter estimation problem in electrical circuit simulation. Both required the use of external simulation codes in the computation of their objective functions. In another application, Mathew et al. [2002] use APPSPACK instead of a gradient-based optimization method to solve a problem in microfluidics. Their choice reflects the fact that portions of their objective function are nonsmooth and that some necessary sensitivity computations are extremely expensive. In a forging process problem, Chiesa et al. [2004] are faced with an objective function based on several stand-alone codes including a mesh generator and a structural analysis code. In this case, gradient-based optimization methods with finite-differences failed to find a solution because there was not enough accuracy in the objective value to compute an approximate gradient; however, APPSPACK was successful. Other uses of APPSPACK include fitting statistical models in image processing [Kupinksi et al. 2003] and

determining the parameters of a wild fire simulator [Croue 2003]. Finally, we note that some work has also been done in comparing APPSPACK and other derivative-free optimization methods. Gray et al. [2003] analyze the performance of APPSPACK and show that it is preferable to simulated annealing for a transmembrane protein structure prediction problem. Liang and Chen [2003] use NEOS[Dolan et al. 2002] to compare APPSPACK to a limited-memory quasi-Newton method for optimal control of a fed-batch fermentation process and concluded that the APPS algorithm is a more powerful tool for stochastic optimization problems. Fowler et al. [2004] compare APPSPACK and six other direct search methods for solving a set of groundwater problems based on the U.S Geological Survey MODFLOW simulator [McDonald and Harbaugh 1988]. The results obtained from this work show that APPSPACK is a highly competitive option for derivative-free optimization.

References

Chiesa, M. L., Jones, R. E., Perano, K. J., and Kolda, T. G. 2004. Parallel optimization of forging proceeses for optimal material properties. Tech. rep., Sandia National Laboratories.

Choi, T. D., Eslinger, O. J., Gilmore, P., Kelley, C. T., and Gablonsky, J. M. July 1999. IFFCO: Implicit filtering for constrained optimization, version 2. Tech. Rep. CRSC-TR99-23, Center for Research in Scientific Computing, North Carolina State University.

Croue, G. 2003. Optimisation par la méthode APPS d'un problème de propagation d'interfaces (in French). M.S. thesis, Ecole Centrale de Lyon, France.

Dolan, E. D., Fourer, R., Moré, J. J., and Munson, T. J. 2002. The NEOS server for optimization: version 4 and beyond. Tech. Rep. MCS-TM-250, Argonne National Laboratory.

Fowler, K. R., Reese, J. P., Kees, C. E., J. E. Dennis, D., Kelley, C. T., Miller, C. T., Audet, C., Booker, A. J., Couture, G., Darwin, R. W., Farthing, M. W., Finkel, D. E., Gablonsky, J. M., Gray, G., and Kolda, T. G. 2004. A comparison of optimization methods for problems involving flow and transport phenomena in saturated subsurface systems. in preparation.

Gilmore, P. and Kelley, C. T. 1995. An implicit filtering algorithm for optimization of functions with many local minima. *SIAM Journal on Optimization 5*, 269–285.

Gray, G. A., Kolda, T. G., Sale, K. L., and Young, M. M. 2003. Optimizing an empirical scoring function for transmembrane protein structure determination. Tech. Rep. SAND2003-8516, Sandia National Laboratories, Livermore, CA 94551. September.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comp. 22*, 789–828.

Gropp, W. D. and Lusk, E. 1996. User's guide for `mpich`, a portable implementation of MPI. Tech. Rep. ANL-96/6, Mathematics and Computer Science Division, Argonne National Laboratory.

Hough, P. D., Kolda, T. G., and Patrick, H. A. 2000. Usage manual for appspack version 2.0. Tech. Rep. SAND2000-8843, Sandia National Laboratories.

Hough, P. D., Kolda, T. G., and Torczon, V. J. 2001. Asynchronous parallel pattern search for nonlinear optimization. *SIAM Journal on Scientific Computing 23,* 1, 134–156.

Kelley, C. 1999. *Iterative Methods for Optimization.* Number 18 in Frontiers in Applied Mathematics. SIAM, Philadelphia, PA.

Kolda, T. G. 2004. Revisiting asynchronous parallel pattern search. Tech. Rep. SAND2004-8055, Sandia National Laboratories, Livermore, CA 94551. February.

Kolda, T. G., Lewis, R. M., and Torczon, V. 2003. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review 45,* 3, 385–482.

Kolda, T. G. and Torczon, V. J. 2003. Understanding asynchronous parallel pattern search. In *High Performance Algorithms and Software for Nonlinear Optimization*, G. Di Pillo and A. Murli, Eds. Applied Optimization, vol. 82. Kluwer Academic Publishers, Boston, 316–335.

Kolda, T. G. and Torczon, V. J. 2004. On the convergence of asynchronous parallel pattern search. *SIAM Journal on Optimization 14,* 4, 939–964.

Kupinksi, M. A., Clarkson, E., Hoppin, J. W., Chen, L., and Barrett, H. H. 2003. Experimental determination of object statistics from noisy images. *J. Opt. Soc. Am. A 20,* 3 (March), 421–429.

Lewis, R. M. and Torczon, V. 1996. Rank ordering and positive bases in pattern search algorithms. Tech. Rep. 96–71, Institute for Computer Applications in Science and Engineering, Mail Stop 132C, NASA Langley Research Center, Hampton, Virginia 23681–2199.

Lewis, R. M., Torczon, V., and Trosset, M. W. 2000. Direct search methods: Then and now. *Journal of Computational and Applied Mathematics 124,* 1–2 (December), 191–207.

Liang, J. and Chen, Y.-Q. 2003. Optimization of a fed-batch fermentation process control competition problem using the NEOS server. *Proceedings of the I MECH E Part I Journal of Systems & Control Engineering 20,* 3 (March), 421–429.

Lucidi, S. and Sciandrone, M. 2002. On the global convergence of derivative-free methods for unconstrained optimization. *SIAM Journal on Optimization 13,* 1, 97–116.

Mathew, G., Petzold, L., and Serban, R. 2002. Computational techniques for quantification and optimization of mixing in microfluidic devices. http://www.engineering.ucsb.edu/˜cse/Files/MixPaper.pdf.

McDonald, M. G. and Harbaugh, A. W. 1988. A modular three dimensional finit difference groundwater flow model. *U.S. Geological Survey Techniques of Water Resources Investigations*.

Sleator, D. and Tajan, R. 1985. Self-adjusting binary search trees. *J. ACM 32*, 652–686.

Torczon, V. 1995. Pattern search methods for nonlinear optimization. *SIAG/OPT Views-and-News: A Forum for the SIAM Activity Group on Optimization 6*, 7–11.

Wright, M. H. 1996. Direct search methods: Once scorned, now respectable. In *Numerical Analysis 1995 (Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis)*, D. F. Griffiths and G. A. Watson, Eds. Pitman Research Notes in Mathematics, vol. 344. CRC Press, 191–208.

Yu, W. 1979. Positive basis and a class of direct search techniques. *Scientia Sinica Special Issue of Mathematics, 1*, 53–67.