Algorithm 856: APPSPACK 4.0: Asynchronous Parallel Pattern Search for Derivative-Free Optimization

GENETHA A. GRAY and TAMARA G. KOLDA
Sandia National Laboratories

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 serially or in parallel, regardless of whether 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.

1. INTRODUCTION

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

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

Here, $f : \mathbb{R}^n \rightarrow \mathbb{R} \cup \{+\infty\}$ 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 Equation (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]. More specifically, APPSPACK version 4.0 is a software implementation of the APPS algorithm presented in Kolda [2004], while previous versions of APPSPACK are based on the algorithm presented in Hough et al. [2001]. It should also be noted that APPSPACK 4.0 is the first version with a related publication.

The APPS algorithm is a variant on generating set search, as described in Kolda et al. [2003]. It is provably convergent if the underlying objective function is suitably smooth [Kolda and Torczon 2003, 2004; Kolda 2004]. Moreover, APPS is a direct search method, thus gradient information is not required. Therefore, it is applicable to a variety of contexts.

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 most closely matching 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 simulator outputs at the same points, depending on input $\mathbf{x}$. Note that in order to discover the $\mathbf{x}$ that yields the best fit for 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 serially or in parallel, regardless of whether the objective function itself runs in parallel.
—The software is freely available under the terms of L-GPL.

These advantages have prompted users to employ APPSPACK for a wide variety of applications (see, e.g., Hough et al. [2001], Mathew et al. [2002], Chiesa et al. [2004], Kupinksi et al. [2003], Croue [2003], Gray et al. [2004], and Flower et al. [2004]).

APPSPACK 4.0 is written in C++ and uses MPI [Gropp et al. 1996; Gropp and Lusk 1996] for parallelism. The underlying APPS algorithm and its algorithm-specific class structures are described in Section 2. Section 3 describes the methods and class structures used by APPSPACK to manage the function evaluation process; these tools are not unique to the APPS algorithm and could in fact be used in other computer software programs. Potential users of APPSPACK will be most interested in Section 4, where we explain how to use APPSPACK, and Section 6, where we summarize and reference some successful applications. In addition, customizations of APPSPACK are discussed in Section 5. Sections 5.1 and 5.2 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 general linear and nonlinear constraints is discussed in Section 5.3. The implementation of general data structures for handling these types of constraints is underway and will be communicated in a future publication and software release.

The notation in this article is as follows. A boldface capital letter, for example, $\mathbf{T}$, denotes a set of vectors. A script capital letter, for example, $\mathcal{I}$, denotes a set of indices. A boldface lowercase letter, for example, $\mathbf{x}$, 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, for example, α, represent scalars.

2. APPS ALGORITHM AND IMPLEMENTATION

The APPS algorithm includes numerous details that are essential for efficient and correct implementation, but not for a 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 $\mathbf{E}$ and see if there exists a point $\mathbf{y}$ such that

$$\mathbf{y} \prec \mathbf{x}.$$

The notation "$\prec$" (described in detail in Figure 2) essentially means $f(\mathbf{y}) < f(\mathbf{x})$. If $\mathbf{E}$ contains such a point $\mathbf{y}$, 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.

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 $\mathcal{I} := \{i \,:\, \Delta_i \geq \Delta_{\text{tol}}$ 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.
    ○ PARENT_TAG$(\mathbf{y})$:= TAG$(\mathbf{x})$
    ○ DIRECTION_INDEX$(\mathbf{y})$:= i
    ○ STEP$(\mathbf{y})$:= Δ_i
- Set τ_i := 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 $\mathbf{z}$ be the point in $\mathbf{E}$ such that $\mathbf{z} \prec \mathbf{y}$ for all other $\mathbf{y} \in \mathbf{E}$.
If $\mathbf{z} \prec \mathbf{x}$, 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 $\Delta_i := \max\{$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 PARENT_TAG$(\mathbf{y}) = $ TAG$(\mathbf{x})$, then let $i = $ 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.

Fig. 1.   APPS algorithm.

A detailed procedural version of APPS is given in Figure 1; for a complete mathematical description and analysis, see Kolda [2004]. The iteration procedure described in Figure 1 can be divided into two parts: steps (1) and (3)–(5), and step (2). The implementation details of the first part are specifically tailored to the APPS algorithm and described in the remainder of this section. The second part includes more general features of the function evaluation management used by APPSPACK and its implementation details are described in the next section.

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

$$\mathbf{y} = \mathbf{x} + \Delta_i \mathbf{d}_i, \tag{3}$$

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*, that is, 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 (3). Once a trial point has been evaluated, its function value $f(\mathbf{y}) \in \mathbb{R} \cup \{+\infty\}$ is also stored in `Point`. All function values are stored as `APPSPACK::Value` objects which 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 become the new best point (i.e. replace the current 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 next.

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

$$f(\mathbf{y}) < f(\mathbf{x}) - \alpha \Delta^2, \tag{4}$$

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 a trial point satisfies Equation (4) with respect to its parent, state information is stored in object `Point`. The state also specifies whether a trial point has been evaluated.

Satisfying decrease condition Equation (4) is only part of the comparison APPSPACK uses to determine whether a point is eligible to replace the current best point. The comparison also considers the corresponding function values as compared to that of the current best point 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.

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($\mathbf{y}$);

Let **y** and **x** be `Point` objects. The following procedure determines if $\mathbf{y} \prec \mathbf{x}$.
- If $\text{TAG}(\mathbf{y}) = \text{TAG}(\mathbf{x})$ (i.e., they are the same point), then $\mathbf{y} \not\prec \mathbf{x}$. Else, continue.
- If **y** does not satisfy (4) with respect to its parent, then $\mathbf{y} \not\prec \mathbf{x}$. Else, continue.
- If **x** does not satisfy (4) with respect to its parent, then $\mathbf{y} \prec \mathbf{x}$ (note that **y** has been evaluated and does satisfy (4) 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 $\text{TAG}(\mathbf{y}) < \text{TAG}(\mathbf{x})$, then $\mathbf{y} \prec \mathbf{x}$. Else $\mathbf{y} \not\prec \mathbf{x}$.

Fig. 2.   Determining if $\mathbf{y} \prec \mathbf{x}$.

—the state information; and

—its function value $f(\mathbf{y})$, if it has been evaluated.

The state information indicates whether the point has been evaluated, and if it has, whether 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

Bound constraints are supported via the `APPSPACK::Constraints::Bounds` class. The constraints object is passed as an argument to the `APPSPACK::Solver`, which executes the APPS algorithm so that we leave open the possibility of allowing user-developed constraint classes in future versions.

The bounds on the variables are specified by the user in the APPSPACK input file (see Section 4.2). 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 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 various important roles such as 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 3.4).

In the case that all bounds are finite, the user may choose either to employ the default scaling vector or to provide one in the APPSPACK input file ("Scaling" in the "Bounds" sublist). The default scaling vector is defined using (finite) 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

$$s_i = u_i - l_i \quad \text{for } i = 1, \ldots, n. \tag{5}$$

If finite bounds are not provided or do not exist, **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, given the current best point $\mathbf{x}$, 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, $\mathbf{e}_i$ is the ith unit vector, and $u_i \in \mathbb{R} \cup \{\infty\}$ and $\ell_i \in \mathbb{R} \cup \{-\infty\}$ are the bounds on x_i (as defined in Equation (1)).

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 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 this 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 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 *pseudostep* must be determined. The pseudostep $\tilde{\Delta}_i$ is the longest possible step that is feasible, 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 step length. This criterion was chosen because it can be shown that if the objective function is continuously differentiable, then the norm of the gradient (or an analogous measure of the constrained measure of stationarity in the bound-constrained case) can be bounded as a multiple of the step size, for example, $\|\nabla f(\mathbf{x})\| = O(\max_i\{\Delta_i\})$ [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; that is, if

$$\Delta_i < \Delta_{\text{tol}} \quad \text{for } i = 1, \ldots, p. \tag{6}$$

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 Equation (5) is used.

An alternative stopping condition is based on whether the function has reached a specified threshold. This criterion 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

$$f(\mathbf{x}) \leq f_{\text{tol}}, \tag{7}$$

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 criterion might be useful when the function evaluations are based on a particularly expensive simulation, and the user wants to adhere to a specified budget for evaluations. By default this stopping criterion is not used, but it can be activated by the user specifying a maximum number of function evaluations in the APPSPACK input file ("Maximum Evaluations" in the "Solver" sublist).

3. APPSPACK FUNCTION EVALUATION MANAGEMENT

This section describes the implementation details and specific C++ objects related to management of the function evaluation process used in APPSPACK version 4.0. Although we describe the evalution conveyor, executor, evaluators, and cache, as they pertain to APPSPACK, it should be noted that they are merely a general way to handle the process of parallelizing multiple independent function evaluations and efficiently balance the computational load. Thus, these classes may be applicable to other software packages.

3.1 The Evaluation Conveyor

From the point of view of the APPS algorithm, the evaluation conveyor simply works as follows: A set of unevaluated points $\mathbf{T}$ is exchanged for a set of evaluated points $\mathbf{E}$ (step (2) of Figure 1). The set $\mathbf{T}$ may be empty. However, because returning an empty set of evaluated points means that the current iteration cannot proceed, the set $\mathbf{E}$ must always be nonempty.

Within the conveyor, a trial point moves through three stages. The first stage is for the trial point 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 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 to follow.

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 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 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**. 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 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 this point (see Section 3.4). 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 3.2. 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
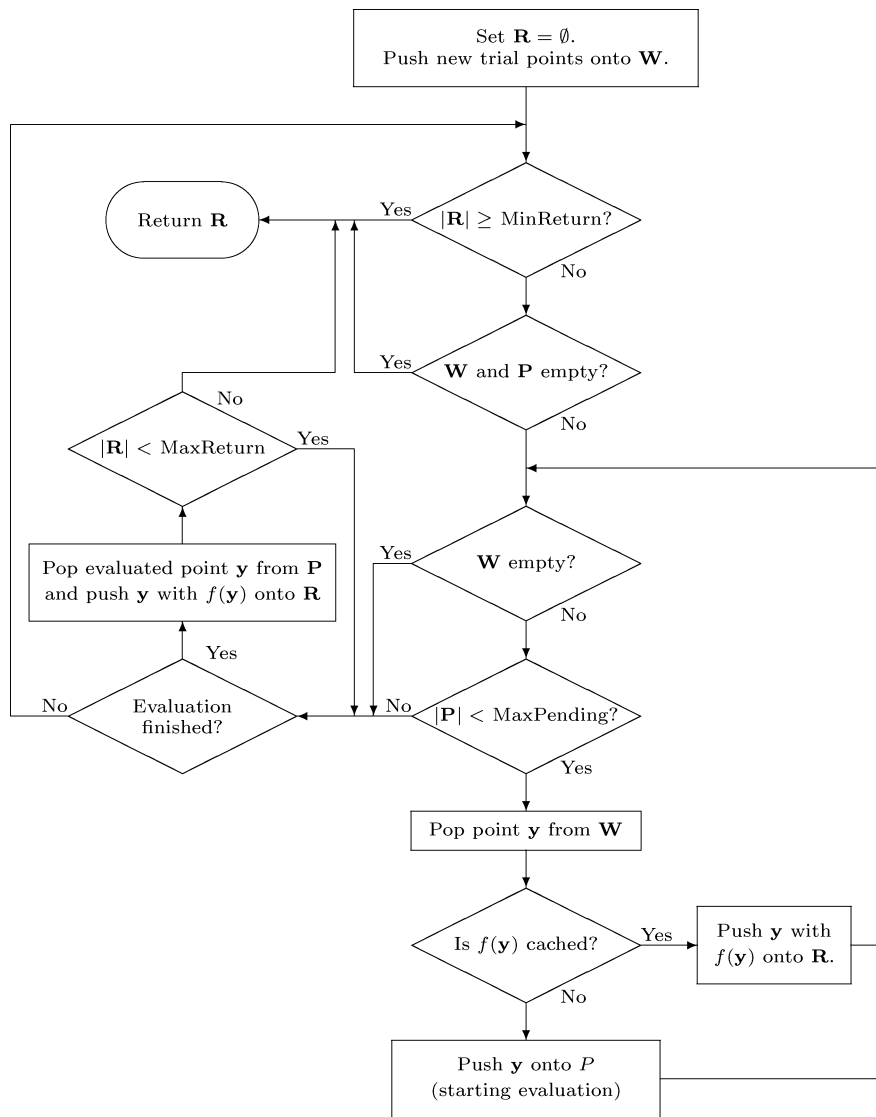
Fig. 3.   Conveyor actions for trial point exchange.

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.

3.2 Executors: Function Evaluation Coordinators

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 **y** and its corresponding tag, TAG(**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 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`, the class that runs the APPS algorithm. APPSPACK 4.0 contains two concrete implementations of the executor, namely, `APPSPACK::Executor::MPI` and `APPSPACK::Executor::Serial`, as well as the corresponding executables, `appspack_mpi` and `appspack_serial`. In addition, the executor can be customized by the user, as explained in Section 5.1.

The MPI executor that comes with APPSPACK spawns function evaluations to worker processes. All processors save one are workers; that is, if `appspack_mpi` is run with five processes, then there will be four workers and the size of the pending queue **P** will be four. Each time APPSPACK calls the spawn function, the executor sends an MPI message to an idle worker, containing both a point to be evaluated and its corresponding tag. The executor marks the worker as busy. The worker performs the evaluation by calling the evaluator (see Section 3.3) and returns an MPI message with the tag, a Boolean indicating if there is a function value, the function value itself, and a message string (e.g., `"Success"`). When APPSPACK calls the receive function, the executor checks for pending messages from a worker (and there may not be any), collects the result, marks the worker that returned the message as idle, and returns the result to APPSPACK.

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 3.3) directly. While there may be some situations where the serial version of APPSPACK proves useful, this mode is provided primarily for testing purposes.

The executor may be customized, the reasons and methods for which are discussed in Section 5.2.
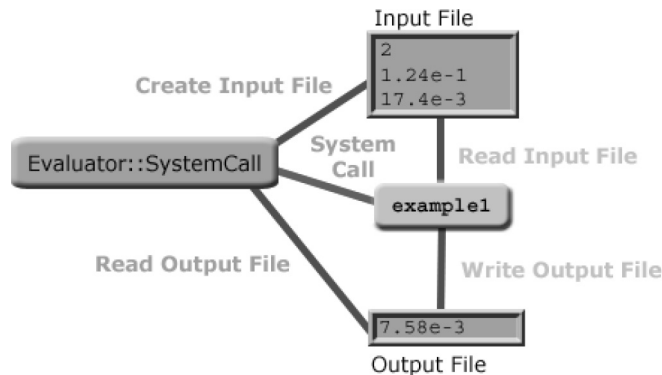
Fig. 4.   The system call evaluator.

3.3 Evaluators

The actual objective function evaluation of the trial points is handled by the evaluator, whose abstract interface is defined in the `APPSPACK::Evaluator::Interface` class. This structure allows the user to either employ the provided default evaluator or create a customized one. The default evaluator is the `APPSPACK::Evaluator::SystemCall` class, which 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 4. Information regarding the user-provided executable and the formats of function input and output files is given in Section 4.1.

The evaluator is its own entity and not part of the executor, although it is related. In MPI mode, each worker possesses its own evaluator object and receives messages from the executor on the master processor with the information (e.g., the point to be evaluated) it needs to pass on to the evaluator. We also note that any information needed to construct and intialize the evaluator has to be passed from the master to the worker processes via an MPI message before the APPSPACK::Solver::solve function is called. In our case, for example, the master sends the workers the name of the executable to be used for the function evaluation. Note that, in serial mode, the executor owns the evaluator directly and calls it in the spawn function.

By default, APPSPACK runs function evaluations as separate executables and communication with the evaluation executable is done via file input and output. In other words, each worker makes an external system call, as illustrated in Figure 5. This default design ensures applicability of APPSPACK to simulation-based optimization because simulations are often too complex to be easily or reasonably encapsulated into a subroutine. Moreover, allowing the user to supply a separate executable extends the usability of APPSPACK, since, for example, the user can write the program for the function evaluation in
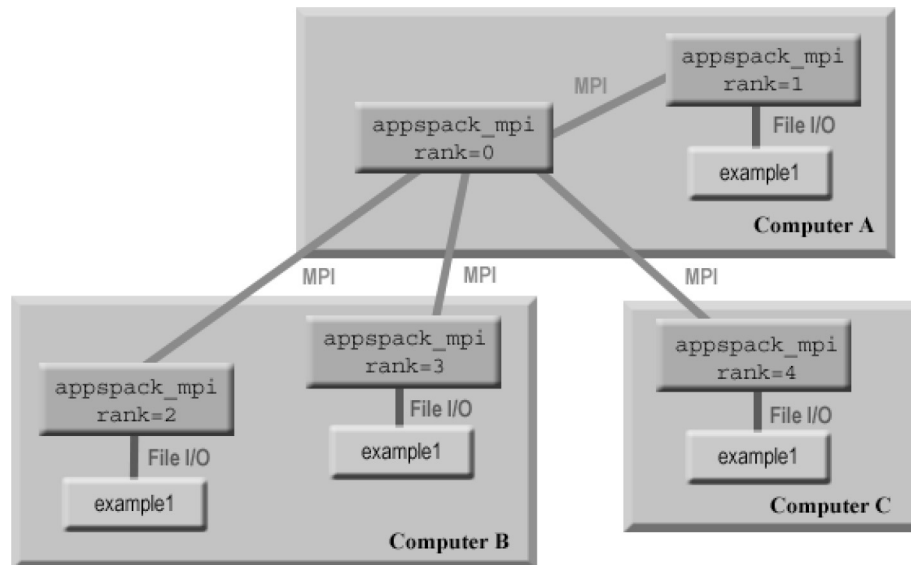
Fig. 5.   Parallel APPSPACK using the (default) system call evaluator.

any language or 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 simulation time often makes the external function calls and file I/O negligible. For applications in which the user may prefer to eliminate this overhead (e.g., the function evaluation may be relatively inexpensive), the user can provide a customized evaluator, as detailed in Section 5.1.

3.4 Cache

Because the APPS algorithm is based on searching a pattern of points that lies on a regular grid, the same point may be revisited several times. Thus, to avoid evaluating the objective function at any given 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 $\mathbf{R}$, the conveyor stores this point and its corresponding function value in the cache. Then, before sending any point to the pending queue $\mathbf{P}$, the conveyor first checks the cache to see if its value has already been calculated. If it has, the cached function value is used instead of repeating the function evaluation.

   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 "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 that include only the vector itself and its corresponding function value. This structure eases storage and comparison, including special rules for point comparison, as explained next.

APPSPACK does not require exact matched in-cache lookups. Rather, it only requires the points to be ε-equal. Specifically, trial points $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are ε-equal if

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

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. Correspondingly, this cache comparison test is also based on the tolerance ε and the scaling vector $\mathbf{s}$. Here, trial point $\mathbf{x}$ is ε-less than $\mathbf{y}$ if there exists an index j such that

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

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 ε-equal, and the jth coordinate of $\mathbf{y}$ is sufficiently greater than the jth coordinate of $\mathbf{x}$.

4. USING APPSPACK

APPSPACK 4.0 is available as a free download under the terms of the GNU Lesser General Public License.[1] Instructions for downloading, compiling, and installing APPSPACK 4.0 can be found in the `README` file included with the software. Several resources are available for APPSPACK users, including online documentation, mailing lists, and a bug-reporting system. The access and use of these tools is described in detail in the HTML documentation.

4.1 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. The APPSPACK evaluator calls this executable repeatedly to evaluate different trial points via the C `system()` command.

---

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

The executable command line should accept three input arguments: an input file name, 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 subsequent line contains 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. For example, an output file containing a function value may contain the single line 7.58e-3, whereas an output file with an error string may contain the single line 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 the length of only one line, and the string "Success" is disallowed as an error string. This feature may be useful for identifying the reasons for a 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, stored in the APPSPACK Point structure, to name related files. The tag is also provided as the third input argument of the executable so as to allow the user to uniquely name any additional files that may be created during an 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. So-called leftover files, that is, temporary files that are not deleted, are frequent sources of error and may cause system disk-space problems.

4.2 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

```
# 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
@@
```

Fig. 6.   A sample APPSPACK input file.

the example file shown in Figure 6. Full instructions on specifying the input file can be found in the HTML documentation under "Using the APPSPACK Executables."

The user has the option of defining any number of input 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 software 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 and may affect the algorithm's speed, but should not affect its convergence. Despite the number of choices offered, we have not encountered any difficulties finding appropriate parameter values for any applications. In fact, default values tend to work well in most cases. The two most critical parameters are "Step Tolerance" and "Scaling" because both play important roles in the algorithm itself. We now discuss the parameters in more detail, including some indication of their overall importance.

4.2.1 *Evaluator Parameters.*   The evalutor, described in Section 3.3, handles the actual evaluation of trial points. Its three parameters (Executable Name, Input Prefix, and Output Prefix) merely indicate the names of files, as described in Section 4.1, and more details can be found in the HTML documentation under "Using the APPSPACK Executables."

4.2.2 *Bounds Parameters.*   As described in Section 2.2, it is critical that the variables be reasonably scaled because no gradient information is available to the APPS algorithm. In APPSPACK, a default scaling vector **s** can only be defined if finite upper and lower bounds are provided for each variable, in which case,

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

where **u** and **l** denote (finite) upper and lower bounds, respectively. If complete upper and lower bounds are not available, then the Scaling must be provided by the user in the input file. The bounds parameters are Lower, Upper, Scaling,

`Is Lower`, and `Is Upper`. Details on specifying these parameters can be found in the HTML documentation under "Bounds Parameters."

4.2.3 *Solver Parameters.* The parameters that are called for in the initialization of the APPS algorithm (outlined in Figure 1) are defined using the parameters in this section.

The most important parameter is `Step Tolerance` because it can be application-specific. For example, many simulators are insensitve to small changes in parameter inputs. Thus, larger step tolerance can be used to reduce the overall number of function evaluations required to reach a solution.

The parameters `Initial X`, `Initial F`, and `Initial Step` control the initialization of the method. Note that if `Initial F` is not provided, then it will not be calculated by the method.

The following parameters are used in the algorithm itself: `Minimum Step`, `Contraction Factor`, and `Sufficient Decrease Factor`.

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

The evaluation conveyor, described in Section 3.1, can be forced to run in synchronous mode via the `Synchronous` parameter. Other parameters that influence the conveyor are: `Max Queue Size`, `Minimum Exchange Return`, and `Maximum Exchange Return`.

There are also some options for how the cache is used. The cache can be saved and read-in again using the `Cache Output File` and `Cache Input File` parameters. The `Cache Comparison Tolerance` determines when two points are ε-equal (with respect to the infinity-norm) and should not be set to more than half the `Step Tolerance`.

Lastly, the following two parameters allow the user to control the amount and format of the APPSPACK output and have no effect on the algorithm. The `Debug` parameter specifies how verbose the output should be on a scale of 1 to 7, with higher values producing more output. The `Precision` indicates the number of digits of precision in the output.

The particulars on specifying any of the parameters discussed in this section can be found in the HTML documentation under "Solver Parameters."

4.3 Running APPSPACK and its output

Once an executable and an input file have been created, the user is ready to run APPSPACK. Instructions and sample execution command lines for both serial and parallel versions can be found in the software `README` file.

For basic testing purposes and to confirm that the software is compiled correctly, three examples are provided in the `${APPSPACK}/examples` directory. All three examples use the simple objective function

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

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 (discussed in more detail in Section 5.3). To run any of the examples, the user should go to the examples directory, and consult the README file. Sample output files (e.g., `example1.out`) include results to assist the user in detecting any installation problems, although the asynchronism means that every run will differ somewhat.

In the remainder of this section, we briefly describe the sample APPSPACK output for Example (1) (see `${APPSPACK}/examples/example1.out`). The output begins with a display of the input parameters. When a default parameter is used, it is always indicated by `[default]`; this is important so that users know what the parameter values are and how they were set.

Subsequent output indicates the algorithm's progress. 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, though more information (such as the set **E** of evaluated trial points) is displayed at higher debugging levels. Note that 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 5.3 for an example.

At the end of the output, the `Final State` indicates why the algorithm terminated. In this case, it was because the stopping criterion based on step length was satisfied. Observe that statistics on the number of function evaluations in total and per processor are also printed.

5. CUSTOMIZING APPSPACK

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

5.1 Customizing the Function Evaluations

As described in Section 3.3, the default version of APPSPACK requires function evaluations to be run as separate executables, and any communication between 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 the associated overhead. For example, some function evaluations may require relatively large amounts of auxiliary data and the use of system calls would require this data to be reread for every evaluation. It may also be the case that the function evaluation is inexpensive in comparison with file I/O requirements.

For these applications, the user can create a customized evaluator class, derived 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 7. Note that, unlike the default evaluator shown in Figure 5, each worker itself actually executes the function evaluation.
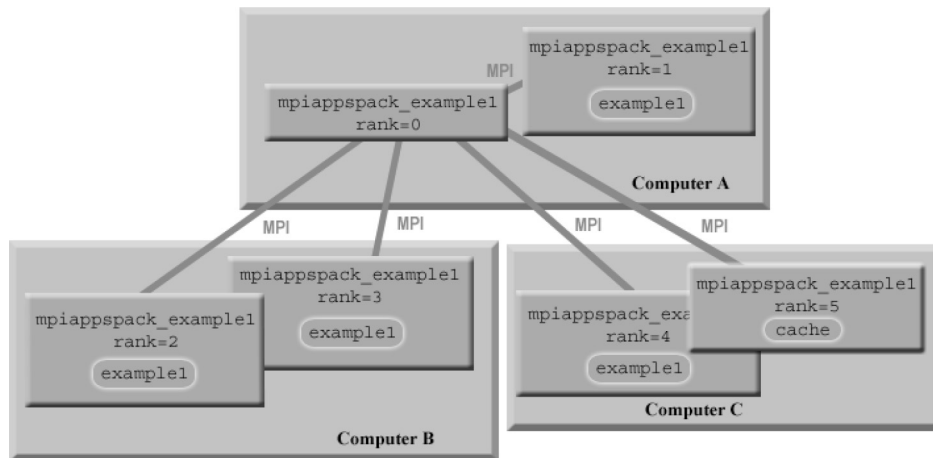
Fig. 7.   Parallel APPSPACK using a customized evaluator.

A documented example of a customized evaluator is provided in the directory ${APPSPACK}/example-custom-evaluator. Additional instructions for creating a custom evaluator are provided in the APPSPACK documentation (see "Customizing APPSPACK" under "Related Pages").

5.2 Customizing the Parallelization

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

Some motivations for customizing the executor are as follows.

—Multilevel parallelism can be achieved by assigning groups of processors to each function evaluation.
—The format of MPI messages sent between the manager and worker can be modified.
—The evaluation of functions can be handed-off to another library, such as a general optimization interface.

A documented example of a customized executor is provided in the directory ${APPSPACK}/example-custom-executor. Additonal instructions for creating a custom evaluator are provided on the APPSPACK documentation (see "Customizing APPSPACK" under "Related Pages").

5.3 General Linear and Nonlinear Constraints

In APPSPACK 4.0, general linear and nonlinear constraints are not handled directly, but can be incorporated via a heuristic barrier approach. We employ the error string capabilities of the function output and a straightforward 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 **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 **y** satisfies the constraint(s).
(2) If **y** is feasible, evaluate the function.
(3) If **y** is infeasible, write the appropriate error string.

Given the results of the function evaluation, either the function value or 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 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 by 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 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 reasonably well for the simple examples here, and we also note that a similar customization of APPSPACK 4.0 has been successful in solving the more complicated constrained optimization problems described by Fowler et al. [2004]. However, this technique is by no means guaranteed to work and, in fact, likely to fail, even in simple cases. Future

versions of APPSPACK will include methods for handling linear and nonlinear constraints.

6. CONCLUSIONS

We have described the underlying algorithm, data structures, features, and advantages 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 operation, 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 user needs.

Many real-world applications have demonstrated the benefits and suitability of APPSPACK. Because these problems are nontrivial and require lengthy descriptions of not only the application of interest, but also related simulators, objective functions, and numerical results, we direct interested readers to the following references.

—In Hough et al. [2001], APPSPACK is successfully applied to a thermal design problem for determining the settings for seven heaters in a thermal deposition furnace, as well as to a 17-variable parameter estimation problem in electrical circuit simulation. In both problems, external simulation codes were required for the computation of the objective function.

—Mathew et al. [2002] use APPSPACK to solve a problem in microfluidics with a nonsmooth objective function and requisite sensitivity computations that are extremely expensive.

—Chiesa et al. [2004] describe a forging process problem in which APPSPACK was successful where gradient-based optimization methods with finite differences failed. In this case, the objective function is based on several stand-alone codes, including a mesh generator and structural analysis code.

—APPSPACK is used to fit statistical models in image processing in Kupinksi et al. [2003].

—Croue [2003] uses APPSPACK to determine the parameters of a wild-fire simulator.

—The performance of APPSPACK is analyzed and shown to be preferable to simulated annealing for a transmembrane protein structure prediction problem in Gray et al. [2004].

—In Liang and Chen [2003], the NEOS optimization server [Dolan et al. 2002] is used to compare APPSPACK to a limited-memory quasi-Newton method for optimal control of a fed-batch fermentation process. The conclusion is that APPS is a more powerful tool for this stochastic optimization problem.

—Fowler et al. [2004] investigate the applicability of a variety of direct search methods to a set of groundwater problems. The objective functions in these problems require the results of the MODFLOW simulator [McDonald and Harbaugh 1988]. The conclusion is that APPSPACK is a highly competitive option.

ACKNOWLEDGMENTS

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. 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 J. Optimization 5*, 269–285.

GRAY, G. A., KOLDA, T. G., SALE, K. L., AND YOUNG, M. M. 2004. Optimizing an empirical scoring function for transmembrane protein structure determination. *INFORMS J. Comp. 16*, 4, 406–418. *Special Issue on Computational Biology*.

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 J. Sci. Comput. 23*, 1, 134–156.

KELLEY, C. T. 1999. *Iterative Methods for Optimization*. Frontiers in Applied Mathematics, no. 18. 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 Rev. 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, Boston, 316–335.

KOLDA, T. G. AND TORCZON, V. J. 2004. On the convergence of asynchronous parallel pattern search. *SIAM J. 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 (Mar.), 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. *J. Comput. Appl. Math. 124*, 1–2 (Dec.), 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 and Control Engineering 20*, 3 (Mar.), 421–429.

LUCIDI, S. AND SCIANDRONE, M. 2002. On the global convergence of derivative-free methods for unconstrained optimization. *SIAM J. 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 finite-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, Boca Raton, FL, 191–208.

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