ASYNCHRONOUS PARALLEL GENERATING SET SEARCH FOR LINEARLY-CONSTRAINED OPTIMIZATION

JOSHUA D. GRIFFIN[†], TAMARA G. KOLDA[†], AND ROBERT MICHAEL LEWIS[*]

Abstract. We describe an asynchronous parallel derivative-free algorithm for linearly-constrained optimization. Generating set search (GSS) is the basis of our method. At each iteration, a GSS algorithm computes a set of search directions and corresponding trial points and then evaluates the objective function value at each trial point. Asynchronous versions of the algorithm have been developed in the unconstrained and bound-constrained cases which allow the iterations to continue (and new trial points to be generated and evaluated) as soon as any other trial point completes. This enables better utilization of parallel resources and a reduction in overall runtime, especially for problems where the objective function takes minutes or hours to compute. For linearly-constrained GSS, the convergence theory requires that the set of search directions conforms to the nearby boundary. This creates an immediate obstacle for asynchronous methods where the definition of nearby is not well defined. In this paper, we develop an asynchronous linearly-constrained GSS method that overcomes this difficulty and maintains the original convergence theory. We describe our implementation in detail, including how to avoid function evaluations by caching function values and using approximate look-ups. We test our implementation on every CUTEr test problem with general linear constraints and up to 1000 variables. Without tuning to individual problems, our implementation was able to solve 95% of the test problems with 10 or fewer variables, 73% of the problems with 11-100 variables, and nearly half of the problems with 100-1000 variables. To the best of our knowledge, these are the best results that have ever been achieved with a derivative-free method for linearly-constrained optimization. Our asynchronous parallel implementation is freely available as part of the APPSPACK software.

Key words. nonlinear programming, constrained optimization, linear constraints, direct search, derivative-free optimization, generalized pattern search (GPS), generating set search (GSS), asynchronous parallel optimization, asynchronous parallel pattern search (APPS)

AMS subject classifications. 90C56, 90C30, 65K05, 15A06, 15A39, 15A48

1. Introduction.

Generating set search (GSS), introduced in [18], is a family of feasible-point methods for derivative-free optimization that encompasses generalized pattern search [31, 2] and related methods. At each iteration, a GSS method evaluates a set of trial points to see if any has a lower function value than the current iterate. This set of evaluations can be performed in parallel, but load balancing is sometimes an issue. For instance, the time for each evaluation may vary or the number of trial points to be evaluated may not be an integer multiple of the number of available processors.

To address the load balancing problem, asynchronous GSS algorithms move to the next iteration as soon as one or more evaluations complete. This permits the parallel processors to exchange evaluated points *immediately* for new trial points, greatly reducing processor idle time. The Asynchronous Parallel Pattern Search PACKage (APPSPACK) was originally developed for pattern search methods for unconstrained problems [15, 22, 21]. As of Version 4 (released in 2004), the underlying algorithm was overhauled to provide better parallelism, implement GSS which generalizes pat-

[†]This work was supported by the Mathematical, Information, and Computational Sciences Program of the U.S. Department of Energy, under contract DE-AC04-94AL85000 with Sandia Corporation. Computational Sciences and Mathematics Research Department, Sandia National Laboratories, Livermore, CA 94551-9159. Email: {jgriffi,tgkolda}@sandia.gov

[*]Department of Mathematics, College of William & Mary, P.O. Box 8795, Williamsburg, Virginia, 23187-8795; buckaroo@math.wm.edu. This research was supported by the Computer Science Research Institute at Sandia National Laboratories and by the National Science Foundation under Grant DMS-0215444.

tern search, and add support for bound constraints [11, 17]. APPSPACK is freely available under the terms of the GNU L-GPL and has proved to be useful in a variety of applications [3, 4, 7, 12, 13, 23, 26, 27, 28, 30]. The software can be run in synchronous or asynchronous mode. In numerical experiments, our experience is that the asynchronous mode has been as fast or faster than the synchronous mode; for example, in recent work, the asynchronous method was 8–30% faster on a collection of benchmark test problems in well-field design [17].

The goal of this paper is to study the problem of handling linear constraints in an asynchronous context. The problem of linear constraints for GSS has been studied by Kolda, Lewis, and Torczon [20], who present a GSS method for linearly-constrained optimization, and Lewis, Shepherd, and Torczon [24], who discuss the specifics of a serial implementation of GSS methods for linearly constrained optimization as well as numerical results for five test problems. Both these papers build upon previous work by Lewis and Torczon [25].

Key to understanding the difficulties encountered when transforming a synchronous GSS algorithm to an asynchronous one is understanding how trial points are produced in both approaches. At each iteration of a synchronous algorithm, a set of search directions is computed; corresponding trial points are then produced by taking a step of fixed length along each direction from the current iterate. Convergence theory requires that the search directions conform to the nearby boundary where the definition of "nearby" depends on the uniform step-length that is used to compute each trial point. In the asynchronous case, however, if a trial point corresponding to a particular direction completes and there is no improvement to the current iterate, a new trial point is generated by taking a reduced step along this same direction. Unfortunately, reducing the step-size can change the definition of the nearby boundary, necessitating a reexamination of the search directions. Thus, to maintain similar convergence properties, an asynchronous algorithm must be able to simultaneously handle multiple definitions of "nearby" when generating search directions. In this paper, our contribution is to show how to handle linear constraints by computing appropriate search directions in an asynchronous context.

The asynchronous algorithm for linearly-constrained GSS is implemented in Version 5 of APPSPACK. The implementation features details that make it suitable for expensive real-world problems: scaling variables, re-use of function values, and nudging trial points to be on exactly the boundary. We also explain how to compute the search directions (relying heavily on [24]), re-use previously computed directions, and strategies for augmenting the search directions.

Our implementation has been tested on both real-world and academic test problems. For instance, researchers at Sandia National Laboratories have used APPSPACK to solve linearly-constrained problems in microfluidics and in engine design. Here we report extensive numerical results on the CUTEr test collection. We consider every problem with general linear constraints and less than 1000 variables. Without tuning to individual problems, our implementation was able to solve 95% of the test problems with 10 or fewer variables, 73% of the problems with 11-100 variables, and nearly half of the problems with 100-1000 variables, including a problem with 505 variables and 2000 linear constraints. To the best of our knowledge, these are the best results that have ever been achieved with a derivative-free method for linearly-constrained optimization. The CUTEr problems have trivial function evaluations; thus, in order to simulate expensive objective function evaluations, we introduce artificial time delays. In this manner, we are able to do computations that reflect our experience with real-world problems and are able to compare the synchronous and asynchronous modes in the software. Our results show that the asynchronous mode was up to 24% faster.

Throughout the paper, the linearly-constrained optimization problem we consider is

$$\begin{array}{rc} \text{minimize} & f(x) \\ \text{subject to} & c_L \leq A_I x \leq c_U \\ & A_E x = b. \end{array} \tag{1.1}$$

Here $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function. The matrix A_I represents the linear inequality constraints, including any bound constraints. Inequality constraints need not be bounded on both sides; that is, we allow for entries of c_L to be $-\infty$, and entries of c_U to be $+\infty$. The matrix A_E represents the equality constraints.

The paper is organized as follows. We describe an asynchronous GSS algorithm for linearly-constrained optimization problems in §2. In §3, we show that this algorithm is guaranteed to converge to a KKT point under mild conditions. Moreover, the asynchronous algorithm has the same theoretical convergence properties as its synchronous counterpart in [20, 24]. Details that help to make the implementation efficient are presented in §4, and we include numerical results on problems from the CUTEr [10] test set in §5. We draw conclusions and discuss future work in §6.

2. Asynchronous GSS for problems with linear constraints. Here we describe the algorithm for parallel, asynchronous GSS for linearly-constrained optimization. Kolda, Lewis, and Torczon [20] outline a GSS algorithm for problems with linear inequality constraints and consider both the simple and sufficient decrease cases. Lewis, Shepherd, and Torczon [24] extend this method to include linear equality constraints as well; if our algorithm required all points to be evaluated at each iteration, it would be equivalent to their method. Kolda [17] describes a parallel asynchronous GSS method for problems that are either unconstrained or bound constrained, considering both the simple and sufficient decrease cases. Here, we revisit the asynchronous algorithm and extend it to handle problems with linear constraints. As much as possible, we have adhered to the notation in [17].

The algorithm is presented in Algorithm 1, along with two subparts in Algorithms 2 and 3. Loosely speaking, each iteration of the algorithm proceeds as follows.

1. Generate new trial points according to the current set of search directions and corresponding step lengths.
2. Submit points to the evaluation queue and collect points whose evaluations are complete.
3. If one of the evaluated trial points sufficiently improves on the current iterate, set it to be the new “best point” and compute a new set of search directions.
4. Otherwise, update the step lengths and compute additional search directions (if any) for the next iteration.

The primary change from Kolda [17] is that now the search directions can change at every iteration. The set of search directions is recomputed every time a new “best point” is discovered and additional directions may be added to the set of search directions as the various step lengths decrease.

2.1. Algorithm notation. In addition to the parameters for the algorithm (discussed in §2.2), we assume that the user provides the linear constraints explicitly and some means for evaluating $f(x)$. The notation in the algorithm is as follows. We let Ω denote the feasible region. Subscripts denote the iteration index.

The vector $x_k \in \Omega \subseteq \mathbb{R}^n$ denotes the *best point*, i.e., the point with the lowest function value at the beginning of iteration k.

The set of *search directions* for iteration k is denoted by $\mathcal{D}_k = \{d_k^{(1)}, \ldots, d_k^{(|\mathcal{D}_k|)}\}$. The superscripts denote the *direction index*, which ranges between 1 and $|\mathcal{D}_k|$ at iteration k. Theoretically, we need only assume that $\| d_k^{(i)} \|$ is uniformly bounded. For simplicity in our discussions and because it matches our implementation, we assume

$$\| d_k^{(i)} \| = 1 \text{ for } i = 1, \ldots, |\mathcal{D}_k|. \tag{2.1}$$

The search directions need to positively span the search space. For the unconstrained problem, a positive basis of $\mathbb{R}^n$ is sufficient; for the linearly-constrained case, we instead need a set of vectors that positively spans the local feasible region. Specifically, we need to find a set of *generators* for the local approximate tangent cone denoted by $T(x_k, \epsilon_k)$. This is critical for handling linear constraints and is disucussed in detail in §2.3. Because the method is asynchronous, each direction has its own *step length*, denoted by

$$\Delta_k^{(i)} \text{ for } i = 1, \ldots, |\mathcal{D}_k|.$$

The set $\mathcal{A}_k \subseteq \{1, \ldots, |\mathcal{D}_k|\}$ is the set of *active indices*, that is, the indices of those directions that have an active trial point in the evaluation queue or that are converged (i.e., $\Delta_k^{(i)} < \Delta_{\text{tol}}$). At iteration k, *trial points* are generated for each $i \notin \mathcal{A}_k$. The trial point corresponding to direction i at iteration k is given by $y = x_k + \tilde{\Delta}_k^{(i)} d_k^{(i)}$ (see Algorithm 2); we say that the point x_k is the *parent* of y. In Algorithm 2, the values of i, x_k, $f(x_k)$, and Δ_k are saved as $\textsc{Direction}(y)$, $\textsc{Parent}(y)$, $\textsc{ParentFx}(y)$, and $\textsc{Step}(y)$, respectively. Conversely, pruning of the evaluation queue in Step 17 means deleting those points that have not yet been evaluated; see §2.5.

In this paper, we focus solely on the sufficient decrease version of GSS. This means that a trial point y must sufficiently improve upon its parent's function value in order to be considered as the next best point. Specifically, it must satisfy

$$f(y) < \textsc{ParentFx}(y) - \rho(\textsc{Step}(y)),$$

where $\rho(\cdot)$ is the forcing function. Algorithm 3 checks this condition, and we assume that the forcing function is

$$\rho(\Delta) = \alpha \Delta^2,$$

where Δ is the step length that was used to produce the trial point, and the multiplicand α is a user-supplied parameter of the algorithm. Other choices for $\rho(\Delta)$ are discussed in §3.2.2.

2.2. Initializing the algorithm. A few comments regarding the initialization of the algorithm are in order. Because GSS is a feasible point method, the initial point x_0 must be feasible. If the given point is not feasible, we first solve a different optimization problem to find a feasible point; see §5.2.

The parameter Δ_{tol} is problem-dependent and plays a major role in determining both the accuracy of the final solution and the number of iterations. Smaller choices of Δ_{tol} yield higher accuracy but the price is a (possibly significant) increase in the number of iterations. If all the variables are scaled to have a range of 1 (see §4.1),

Algorithm 1 Asynchronous GSS for linearly-constrained optimization

Require: $x_0 \in \Omega$ ▷ initial starting point
Require: $\Delta_{\text{tol}} > 0$ ▷ step length convergence tolerance
Require: $\Delta_{\min} > \Delta_{\text{tol}}$ ▷ minimum first step length for a new best point
Require: $\delta_0 > \Delta_{\text{tol}}$ ▷ initial step length
Require: $\epsilon_{\max} > \Delta_{\text{tol}}$ ▷ maximum distance for considering constraints nearby
Require: $q_{\max} \geq 0$ ▷ max queue size after pruning
Require: $\alpha > 0$ ▷ sufficient decrease parameter, used in Alg. 3

$\mathcal{G}_0 \leftarrow$ generators for $T(x_0, \epsilon_0)$ where $\epsilon_0 = \min\{\delta_0, \epsilon_{\max}\}$
$\mathcal{D}_0 \leftarrow$ a set containing $\mathcal{G}_0$
$\Delta_0^{(i)} \leftarrow \delta_0$ for $i = 1, \ldots, |\mathcal{D}_0|$
$\mathcal{A}_0 \leftarrow \emptyset$
for $k = 0, 1, \ldots$ **do**
    $\mathcal{X}_k \leftarrow \{\ x_k + \tilde{\Delta}_k^{(i)} d_k^{(i)} \ \mid\ 1 \leq i \leq |\mathcal{D}_k|,\ \ i \notin \mathcal{A}_k\ \}$ (see Alg. 2) ▷ generate trial points
    send trial points $\mathcal{X}_k$ (if any) to the evaluation queue
    collect a (non-empty) set $\mathcal{Y}_k$ of evaluated trial points
    $\bar{\mathcal{Y}}_k \leftarrow$ subset of $\mathcal{Y}_k$ that has sufficient decrease (see Alg. 3)
    if there exists a trial point $y_k \in \bar{\mathcal{Y}}_k$ such that $f(y_k) < f(x_k)$ **then** ▷ successful
        $x_{k+1} \leftarrow y_k$
        $\delta_{k+1} \leftarrow \max\{\textsc{Step}(y_k), \Delta_{\min}\}$
        $\mathcal{G}_{k+1} \leftarrow$ generators for $T(x_{k+1}, \epsilon_{k+1})$ where $\epsilon_{k+1} = \min\{\delta_{k+1}, \epsilon_{\max}\}$
        $\mathcal{D}_{k+1} \leftarrow$ a set containing $\mathcal{G}_{k+1}$
        $\Delta_{k+1}^{(i)} \leftarrow \delta_{k+1}$ for $i = 1, \ldots, |\mathcal{D}_{k+1}|$
        $\mathcal{A}_{k+1} \leftarrow \emptyset$
        prune the evaluation queue to $q_{\max}$ or fewer entries
    else ▷ unsuccessful
        $x_{k+1} \leftarrow x_k$
        $\mathcal{I}_k \leftarrow \{\textsc{Direction}(y) : y \in \mathcal{Y}_k$ and $\textsc{Parent}(y) = x_k\}$
        $\delta_{k+1} \leftarrow \min \left\{ \ \tfrac{1}{2}\Delta_k^{(i)} \ \mid\ i \in \mathcal{I}_k \ \right\} \cup \left\{ \ \Delta_k^{(i)} \ \mid\ i \notin \mathcal{I}_k \ \right\}$
        $\mathcal{G}_{k+1} \leftarrow$ generators for $T(x_{k+1}, \epsilon_{k+1})$ where $\epsilon_{k+1} = \min\{\delta_{k+1}, \epsilon_{\max}\}$
        $\mathcal{D}_{k+1} \leftarrow$ a set containing $\mathcal{D}_k \cup \mathcal{G}_{k+1}$
        $\Delta_{k+1}^{(i)} \leftarrow \begin{cases} \tfrac{1}{2}\Delta_k^{(i)} & \text{for } 1 \leq i \leq |\mathcal{D}_k| \text{ and } i \in \mathcal{I}_k \\ \Delta_k^{(i)} & \text{for } 1 \leq i \leq |\mathcal{D}_k| \text{ and } i \notin \mathcal{I}_k \\ \delta_{k+1} & \text{for } |\mathcal{D}_k| < i \leq |\mathcal{D}_{k+1}| \end{cases}$
        $\mathcal{A}_{k+1} \leftarrow \{\ i \ \mid\ 1 \leq i \leq |\mathcal{D}_k|, i \notin \mathcal{I}_k\ \} \cup \{\ i \ \mid\ 1 \leq i \leq |\mathcal{D}_{k+1}|, \Delta_k^{(i)} < \Delta_{\text{tol}}\ \}$
    end if
    if $\Delta_{k+1}^{(i)} < \Delta_{\text{tol}}$ for $i = 1, \ldots, |\mathcal{D}_{k+1}|$ **then terminate**
end for

choosing $\Delta_{\text{tol}} = 0.01$ means that the algorithm terminates when the change in each parameter is less than 1%.

The minimum step size following a successful iteration must be set to some value greater than Δ_{tol} and defaults to $\Delta_{\min} = 2\Delta_{\text{tol}}$. A typical choice for the initial step length is $\delta_0 = 1$; relatively speaking, bigger initial step lengths are better than smaller ones. The parameter $\epsilon_{\max}$ forms an upper bound on the maximum distance used to determine whether a constraint is nearby and must also be greater than Δ_{tol}. A typical choice is $\epsilon_{\max} = 2\Delta_{\text{tol}}$. The pruning parameter $q_{\max}$ is usually set equal to the number of worker processors, implying that the evaluation queue is always emptied save for points currently being evaluated. The sufficient decrease parameter

Algorithm 2 Generating trial points

for all $i \in \{1, \ldots, |\mathcal{D}_k|\} \setminus \mathcal{A}_k$ **do**
  $\bar{\Delta} = \max\{\ \Delta > 0 \ \mid \ x_k + \Delta d_k^{(i)} \in \Omega\ \}$ ▷ max feasible step
  $\tilde{\Delta}_k^{(i)} = \min\{\Delta_k^{(i)}, \bar{\Delta}\}$
  if $\tilde{\Delta}_k^{(i)} > 0$ **then**
    $y \leftarrow x_k + \tilde{\Delta}_k^{(i)} d_k^{(i)}$
    Step$(y) \leftarrow \Delta_k^{(i)}$
    Parent$(y) \leftarrow x_k$
    ParentFx$(y) \leftarrow f(x_k)$
    Direction$(y) \leftarrow i$
    add y to collection of trial points
  else
    $\Delta_k^{(i)} \leftarrow 0$
  end if
end for

Algorithm 3 Sufficient decrease check

$\bar{\mathcal{Y}}_k \leftarrow \emptyset$
for all $y \in \mathcal{Y}_k$ **do**
  $\hat{f} \leftarrow$ ParentFx(y)
  $\hat{\Delta} \leftarrow$ Step(y)
  if $f(y) < \hat{f} - \alpha\hat{\Delta}^2$ **then**
    $\bar{\mathcal{Y}}_k \leftarrow \bar{\mathcal{Y}}_k \cup \{y\}$
  end if
end for

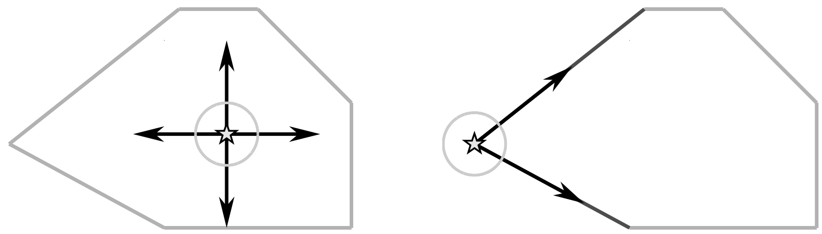
α is typically chosen to be some small constant such as $\alpha = 0.01$.

2.3. Updating the search directions. In Steps 1, 13, and 22, a set of conforming search directions, with respect to x and ϵ, is computed. We seek directions that generate $T(x_k, \epsilon_k)$, the ϵ_k-tangent cone about x_k. Readers unfamiliar with cones and generators may jump ahead to §3.1.1. Several examples of conforming search directions for particular choices of x_k and ϵ_k are shown in Figure 2.1. The idea is to be able to walk parallel to the nearby boundary. The details of actually computing the generators of a particular cone are described in §4.4. The question here is how we define "nearby", i.e., the choice of ϵ_k. The choice of ϵ_k depends on Δ_k; specifically, we set $\epsilon_k = \min\{\Delta_k, \epsilon_{\max}\}$ as is standard [20, 24]. Using an upper bound $\epsilon_{\max}$ prevents us from considering constraints that are too far away. If ϵ_k is too large, the ϵ_k-tangent cone may be empty as seen in Figure 2.1(d).

In the asynchronous case, meanwhile, every search direction has its own step length, $\Delta_k^{(i)}$. Consequently, $\mathcal{D}_k$, the set of search directions at iteration k, must contain generators for each of the following cones:

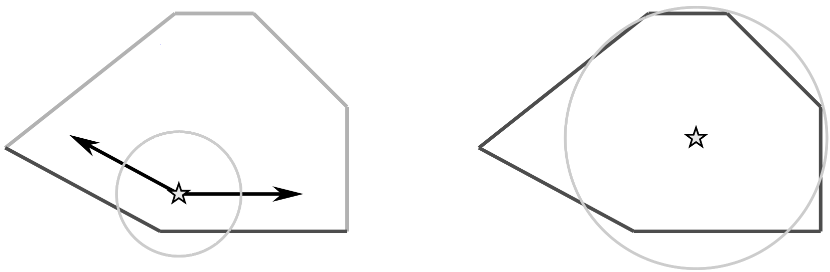
$$T(x_k, \epsilon_k^{(i)}) \text{ where } \epsilon_k^{(i)} = \min\{\Delta_k^{(i)}, \epsilon_{\max}\} \text{ for } i = 1, \ldots, |\mathcal{D}_k|. \tag{2.2}$$

This requirement is not as onerous as it may at first seem. After successful iterations, the step sizes are all equal, so only one tangent cone is relevant (Step 13). It is only after an unsuccessful iteration that generators for multiple tangent cones may



(a) The ϵ-ball does not intersect any constraints; any positive spanning set can be used.

(b) The current iterate is on the boundary and its ϵ-ball intersects with two constraints.



(c) The current iterate is not on the boundary but its ϵ-ball intersects with two constraints.

(d) The value of ϵ is so large that the corresponding ϵ-tangent cone is empty.

FIG. 2.1. *Different sets of conforming directions as x (denoted by a star) and ϵ vary.*

be needed simultaneously. As the individual step sizes $\Delta_k^{(i)}$ are reduced in Step 24, generators for multiple values of ϵ may need to be included. Because $\epsilon_{k+1} \in \{\epsilon_k, \frac{1}{2}\epsilon_k\}$ in Step 21, we need add *at most one* set of search directions per iteration in order to satisfy (2.2). If $\delta_{k+1} = \delta_k$ or $\delta_{k+1} \geq \epsilon_{\max}$, then $\epsilon_{k+1} = \epsilon_k$, so there will be no difference between $T(x_{k+1}, \epsilon_{k+1})$ and $T(x_k, \epsilon_k)$. Consequently, we can skip the calculation of extra directions in Step 13 and Step 22. Even when $\epsilon_{k+1} < \epsilon_k$, the corresponding set of new ϵ-active constraints may remain unchanged, i.e. $N(x_{k+1}, \epsilon_{k+1}) = N(x_k, \epsilon_k)$ implying $\mathcal{D}_{k+1} = \mathcal{D}_k$. When the ϵ-active constraints do differ, we generate conforming directions for the new tangent cone in Step 22 and merge the new directions with the current direction set in Step 23.

2.4. Trial Points. In Step 6, trial points are generated for each direction that does not already have an associated trial point and is not converged. Algorithm 2 provides the details of generating trial points. If a full step is not possible, then the method takes the longest possible feasible step. However, if no feasible step may be taken in direction $d_k^{(i)}$, the step length $\Delta_k^{(i)}$ is set to zero. Note that STEP(y) stores $\Delta_k^{(i)}$ as opposed to the truncated step size $\tilde{\Delta}_k^{(i)}$. This is important because the stored step is used as the basis for the new initial step in Step 12 and so prevents the steps from becoming prematurely small even if the feasible steps are short.

The set of trial points collected in Step 8 may not include all the points in $\mathcal{X}_k$ and may include points from previous iterations.

2.5. Successful Iterations. The candidates for the new best point are first restricted (in Step 9) to those points that satisfy the sufficient decrease condition. The sufficient decrease condition is with respect to the point's parent, which is not necessarily x_k. The details for verifying this condition are in Algorithm 3. Next, in Step 10, we check whether or not any point strictly improves the current best function value. If so, the iteration is called *successful*.

In this case, we update the best point, reset the search directions and corresponding step lengths, prune the evaluation queue, and reset the set of active directions $\mathcal{A}_{k+1}$ to the empty set. Note that we reset the step length to δ_{k+1} in Step 15 and that this value is the maximum of $\Delta_{\min}$ and the step that produced the new best point (see Step 12). The constant $\Delta_{\min}$ is used to reset the step length for each new best point and is needed for the theory that follows; see Proposition 3.8. In a sense, $\Delta_{\min}$ can be thought of as a mechanism for increasing the step size, effectively expanding the search radius after successful iterations.

The pruning in Step 17 ensures that the number of items in the evaluation queue is always finitely bounded. In theory, the number of items in the queue may grow without bound [17].

2.6. Unsuccessful Iterations. If the condition in Step 10 is not satisfied, then we call the iteration unsuccessful. In this case, the best point is unchanged ($x_{k+1} = x_k$). The set $\mathcal{I}_k$ in Step 20 is the set of direction indices for those evaluated trial points that have x_k as their parent. If $\mathcal{I}_k = \emptyset$ (in the case that no evaluated point has x_k as its parent), then nothing changes; that is, $\mathcal{D}_{k+1} \leftarrow \mathcal{D}_k$, $\Delta_{k+1}^{(i)} \leftarrow \Delta_k^{(i)}$ for $i \leftarrow 1, \ldots, |\mathcal{D}_{k+1}|$, and $\mathcal{A}_{k+1} \leftarrow \mathcal{A}_k$. If $\mathcal{I}_k \neq \emptyset$, we reduce step sizes corresponding to indices in $\mathcal{I}_k$ and add new directions to $\mathcal{D}_{k+1}$ as described in §2.3.

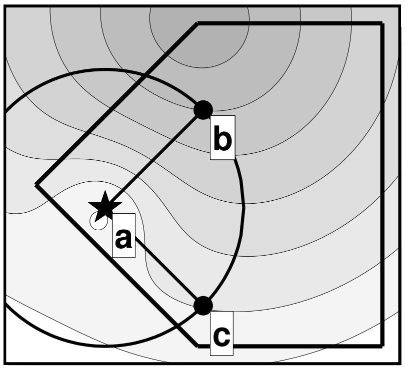
It is important that points never be pruned during unsuccessful iterations. Pruning on successful iterations offers the benefit of freeing up the evaluation queue so that points nearest the new best point may be evaluated first. In contrast, at unsuccessful iterations, until a point has been evaluated, no information exists to suggest that reducing the step size and resubmitting will be beneficial. Theoretically, the basis for Proposition 3.8 hinges upon the property that points are never pruned until a new best point is found.

2.7. An illustrated example. In Figure 2.2, we illustrate six iterations of Algorithm 1, applied to the test problem

$$\begin{array}{lrcl} \text{minimize} & f(x) = \multicolumn{3}{l}{\sqrt{9x_1^2 + (3x_2 - 5)^2} - 5\exp\left(\frac{-1}{(3x_1+2)^2+(3x_2-1)^2+1}\right)} \\ & & 3x_1 & \leq 4 \\ \text{subject to} & -2 \leq & 3x_2 & \leq 5 \\ & & -3x_1 - 3x_2 & \leq 2 \\ & & -3x_1 + 3x_2 & \leq 5. \end{array} \tag{2.3}$$

We initialize Algorithm 1 with $x_0 = \mathbf{a}$, $\Delta_{\text{tol}} = 0.01$ (though not relevant in the iterations we show here), $\Delta_{\min} = 0.02$ (likewise), $\Delta_0 = 1$, $\epsilon_{\max} = 1$, $q_{\max} = 2$, and $\alpha = 0.01$.

The initial iteration is shown in Figure 2.2a. Shaded level curves illustrate the value of the objective function, with darker shades representing lower values. The feasible region is inside of the pentagon. The current best point, $x_0 = \mathbf{a}$, is denoted by a star. We calculate search directions (shown as lines emanating from the current best point to corresponding trial points) that conform to the constraints captured in the ϵ_0-ball. We generate the trial points $\mathbf{b}$ and $\mathbf{c}$, both of which are submitted to the

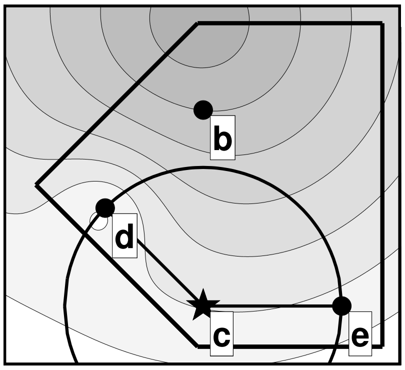


$x_0 = \mathbf{a}$, $\delta_0 = 1$, $\epsilon_0 = 1$
$\mathcal{D}_0 = \left\{ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \right\}$
$\Delta_0^{(1)} = \Delta_0^{(2)} = 1$
$\mathcal{X}_0 = \{\mathbf{b}, \mathbf{c}\}$, Queue $= \{\mathbf{b}, \mathbf{c}\}$

Wait for evaluator to return...

$\mathcal{Y}_0 = \{\mathbf{c}\}$, Queue $= \{\mathbf{b}\}$
$f(\mathbf{c}) < f(\mathbf{a}) - \rho(\Delta_0^{(1)})$
$\Rightarrow$ Successful

FIG. 2.2A. *Iteration $k = 0$ for example problem*

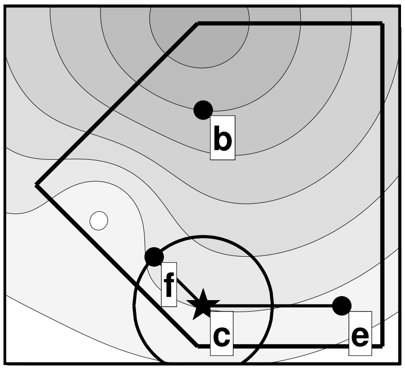


$x_1 = \mathbf{c}$, $\delta_1 = 1$, $\epsilon_1 = 1$
$\mathcal{D}_1 = \left\{ \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}$
$\Delta_1^{(1)} = \Delta_1^{(2)} = 1$
$\mathcal{X}_1 = \{\mathbf{d}, \mathbf{e}\}$, Queue $= \{\mathbf{b}, \mathbf{d}, \mathbf{e}\}$

Wait for evaluator to return...

$\mathcal{Y}_1 = \{\mathbf{d}\}$, Queue $= \{\mathbf{b}, \mathbf{e}\}$
$f(\mathbf{d}) \geq f(\mathbf{c})$
$\Rightarrow$ Unsuccessful

FIG. 2.2B. *Iteration $k = 1$ for example problem*



$x_2 = \mathbf{c}$, $\delta_2 = \frac{1}{2}$, $\epsilon_2 = \frac{1}{2}$
$\mathcal{D}_2 = \mathcal{D}_1$
$\Delta_2^{(1)} = \frac{1}{2}$, $\Delta_2^{(2)} = 1$
$\mathcal{X}_2 = \{\mathbf{f}\}$, Queue $= \{\mathbf{b}, \mathbf{e}, \mathbf{f}\}$

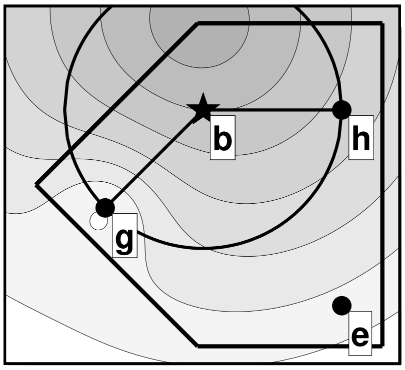
Wait for evaluator to return...

$\mathcal{Y}_2 = \{\mathbf{f}, \mathbf{b}\}$, Queue $= \{\mathbf{e}\}$
$f(\mathbf{b}) < f(\mathbf{a}) - \rho(\Delta_0^{(1)})$ and $f(\mathbf{b}) < f(\mathbf{c})$
$\Rightarrow$ Successful

FIG. 2.2C. *Iteration $k = 2$ for example problem*

evaluation queue. We assume that only a single point, $\mathbf{c}$, is returned by the evaluator. In this case, the point satisfies sufficient decrease with respect to its parent, $\mathbf{a}$, and necessarily also satisfies simple decrease with respect to the current best point, $\mathbf{a}$.

Figure 2.2b shows the next iteration. The best point is updated to $x_1 = \mathbf{c}$. The point $\mathbf{b}$ is what we call an orphan because its parent is not the current best point. The set of nearby constraints changes, so the search directions also change, as shown. The step lengths are all set to $\delta_1 = 1$, generating the new trial points $\mathbf{d}$ and $\mathbf{e}$, which are



$x_3 = \mathbf{b}$, $\delta_3 = 1$, $\epsilon_3 = 1$

$\mathcal{D}_3 = \left\{ \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}$

$\Delta_3^{(1)} = \Delta_3^{(2)} = 1$

$\mathcal{X}_3 = \{\mathbf{g}, \mathbf{h}\}$, Queue $= \{\mathbf{e}, \mathbf{g}, \mathbf{h}\}$

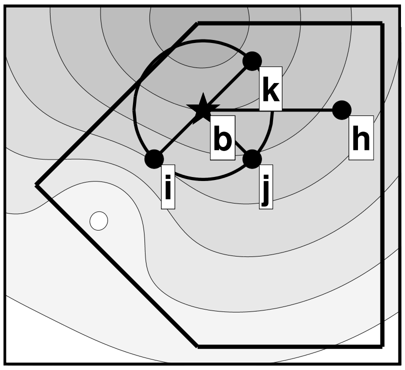
Wait for evaluator to return...

$\mathcal{Y}_3 = \{\mathbf{e}, \mathbf{g}\}$, Queue $= \{\mathbf{h}\}$

$f(\mathbf{g}), f(\mathbf{e}) \geq f(\mathbf{b})$

$\Rightarrow$ Unsuccessful

FIG. 2.2D. *Iteration $k = 3$ for example problem*



$x_4 = \mathbf{b}$, $\delta_4 = \frac{1}{2}$ $\epsilon_4 = \frac{1}{2}$

$\mathcal{D}_4 = \left\{ \mathcal{D}_3, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \right\}$

$\Delta_4^{(i)} = \frac{1}{2}$ for $i = 1, 3, 4$, $\Delta_4^{(2)} = 1$

$\mathcal{X}_4 = \{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$, Queue $= \{\mathbf{h}, \mathbf{i}, \mathbf{j}, \mathbf{k}\}$

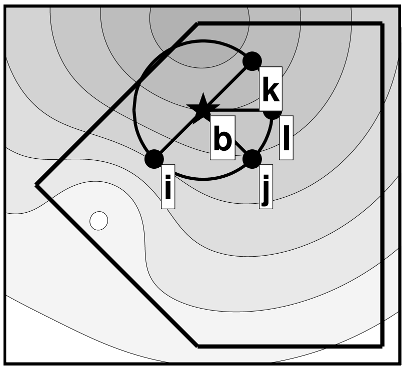
Wait for evaluator to return...

$\mathcal{Y}_4 = \{\mathbf{h}\}$, Queue $= \{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$

$f(\mathbf{h}) \geq f(\mathbf{b})$

$\Rightarrow$ Unsuccessful

FIG. 2.2E. *Iteration $k = 4$ for example problem*



$x_5 = \mathbf{b}$, $\delta_5 = \frac{1}{2}$, $\epsilon_5 = \frac{1}{2}$

$\mathcal{D}_5 = \mathcal{D}_4$

$\Delta_5^{(i)} = \frac{1}{2}$ for $i = 1, 2, 3, 4$

And the process continues...

FIG. 2.2F. *Iteration $k = 5$ for example problem*

submitted to the evaluation queue. Once again, the evaluator returns a single point, $\mathbf{d}$. In this case, $\mathbf{d}$ does not satisfy the sufficient decrease condition, so the iteration is unsuccessful.

In Figure 2.2c, the best point is unchanged, i.e., $x_2 = x_1 = \mathbf{c}$. The value of δ_2 and hence ϵ_2 is reduced to $\frac{1}{2}$. In this case, however, the set of ϵ-active constraints is unchanged, so $\mathcal{D}_2 = \mathcal{D}_1$. The step length corresponding to the first direction, $\Delta_2^{(1)}$, is reduced and a new trial point, **f**, is submitted to the queue. This time, two points return as evaluated, **f** and **b**, the latter of which has the lower function value. In this case, we check that **b** satisfies sufficient decrease with respect to its parent, **a**, and that it also satisfies simple decrease with respect to the current best point, **c**. Both checks are satisfied, so the iteration is successful.

In Figure 2.2d, we have a new best point, $x_3 = \mathbf{b}$. The value of δ_3 is set to 1.0, the step length that was used to generate the point **b**. Conforming search directions are generated for the new ϵ-active constraints. The trial points $\{\mathbf{g}, \mathbf{h}\}$ are submitted to the evaluation queue. In this case, the points **e** and **g** are returned, but neither satisfies sufficient decrease with respect to its parent. Thus, the iteration is unsuccessful.

In Figure 2.2e, the best point is unchanged, so $x_4 = x_3 = \mathbf{b}$. However, though our current point did not change, $\epsilon_4 = \frac{1}{2}$ is reduced because $\delta_4 = \frac{1}{2}$ is reduced. In contrast to iteration 2, the ϵ-active constraints have changed. The generators used for $T(x_4, \frac{1}{2})$ are

$$\left\{ \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \right\}.$$

The first direction is already in $\mathcal{D}_3$; thus, we need only add the last two directions to form $\mathcal{D}_4$. In this iteration, only the point **h** is returned, but it does not improve the function value, so the iteration is unsuccessful.

For Figure 2.2f, we have $\delta_5 = \delta_4$, so there is no change in the search directions. The only change is that the step corresponding to direction 2 is reduced. And the iterations continue.

3. Theoretical properties. In this section we prove global convergence for the asynchronous GSS algorithm described in Algorithm 1. A key theoretical difference between GSS and asynchronous GSS is that all the trial points generated at iteration k may not be evaluated in that same iteration. This necessitates having multiple sets of directions in $\mathcal{D}_k$ corresponding to different ϵ-tangent cones.

3.1. Definitions and terminology.

3.1.1. ϵ-normal and ϵ-tangent cones. Integral to GSS convergence theory are the concepts of tangent and normal cones [20]. A *cone* K is a set in $\mathbb{R}^n$ that is closed under nonnegative scalar multiplication; that is, $\alpha x \in K$ if $\alpha \geq 0$ and $x \in K$. The polar of a cone K, denoted by K°, is defined by

$$K^\circ = \left\{ \, w \;\mid\; w^T v \leq 0 \;\forall\, v \in K \, \right\},$$

and is itself a cone. Given a convex cone K and any vector v, there is a unique closest point of K to v called the projection of v onto K and denoted v_K. Given a vector v and a convex cone K, there exists an orthogonal decomposition such that $v = v_K + v_{K^\circ}$, $v_K^T v_{K^\circ} = 0$, with $v_K \in K$ and $v_{K^\circ} \in K^\circ$. A set $\mathcal{G}$ is said to generate a cone K if K is the set of all nonnegative combinations of vectors in $\mathcal{G}$.

For a given x, we are interested in the ϵ-tangent cone, which is the tangent cone of the nearby constraints. Following [24], we define the ϵ-normal cone $N(x, \epsilon)$ to be the cone generated by the outward pointing normals of constraints within distance ϵ of x. Moreover, distance is measure within the nullspace of A_E. The ϵ-tangent cone is then defined as the polar to the ϵ-normal cone, i.e., $T(x, \epsilon) \equiv N(x, \epsilon)^\circ$.

We can form generators for $N(x, \epsilon)$ explicitly from the rows of A_I and A_E as follows. Let $(A_I)_i$ denote the ith row of A_I and let $(A_I)_\mathcal{S}$ denote the submatrix of

A_I with rows specified by $\mathcal{S}$. Let Z denotes an orthonormal basis for the nullspace of A_E. For a given x and ϵ we can then define the index sets of ϵ-active constraints for A_I as

$$\mathcal{E}_B = \{i : \,|\,(A_I)_i x - (c_U)_i\,| \le \epsilon \|\,(A_I)_i Z\,\| \text{ and } |\,(A_I)_i x - (c_L)_i\,| \le \epsilon \|\,(A_I)_i Z\,\|\} \text{ (both)},$$
$$\mathcal{E}_U = \{i : \,|\,(A_I)_i x - (c_U)_i\,| \le \epsilon \|\,(A_I)_i Z\,\|\} \setminus \mathcal{E}_B \text{ (only upper), and}$$
$$\mathcal{E}_L = \{i : \,|\,(A_I)_i x - (c_L)_i\,| \le \epsilon \|\,(A_I)_i Z\,\|\} \setminus \mathcal{E}_B \text{ (only lower),}$$

and define matrices V_P and V_L as

$$V_P = \begin{bmatrix} (A_I)_{\mathcal{E}_U} \\ -(A_I)_{\mathcal{E}_L} \end{bmatrix}^T \text{ and } V_L = \begin{bmatrix} A_E \\ (A_I)_{\mathcal{E}_B} \end{bmatrix}^T. \tag{3.1}$$

Then the set

$$\mathcal{V}(x, \epsilon) = \{\ v \ \mid\ v \text{ is a column of } [V_P, V_L, -V_L]\ \}$$

generates the cone $N(x, \epsilon)$. We delay the description of how to form generators for the polar $T(x, \epsilon)$ until §4.4 because the details of its construction are not necessary for the theory.

The following measure of the quality of a given set of generators $\mathcal{G}$ will be needed in the analysis that follows and comes from [20, 24, 18]. For any finite set of vectors $\mathcal{G}$, we define

$$\kappa(\mathcal{G}) \equiv \inf_{\substack{v \in \mathbb{R}^n \\ v_K \neq 0}} \max_{d \in \mathcal{G}} \frac{v^T d}{\|\,v_K\,\|\|\,d\,\|}, \text{ where } K \text{ is the cone generated by } \mathcal{G}. \tag{3.2}$$

Examples of $\kappa(\mathcal{G})$ can be found in [18, §3.4.1]. It can be shown that $\kappa(\mathcal{G}) > 0$ if $\mathcal{G} \neq \{0\}$ [20, 25]. As in [20] we make use of the following definition:

$$\nu_{\min} = \min\{\kappa(\mathcal{V}) : \mathcal{V} = \mathcal{V}(x, \epsilon),\ x \in \Omega,\ \epsilon \ge 0,\ \mathcal{V}(x, \epsilon) \neq 0\}, \tag{3.3}$$

which provides a measure of the quality of the constraint normals serving as generators for their respective ϵ-normal cones. Because only a finite number of constraints exists, there is a finite number of possible normal cones. Since $\kappa(\mathcal{V}) > 0$ for each normal cone, we must have that $\nu_{\min} > 0$. We will need the following proposition in the analysis that follows:

Proposition 3.1 ([20]). *If $x \in \Omega$, then for all $\epsilon \ge 0$ and $v \in \mathbb{R}^n$,*

$$\max_{\substack{x+w \in \Omega \\ \|w\|=1}} w^T v \le \|v_{T(x,\epsilon)}\| + \frac{\epsilon}{\nu_{\min}} \|v_{N(x,\epsilon)}\|,$$

where $\nu_{\min}$ is defined in (3.3).

3.1.2. A measure of stationarity. In our analysis, we use the first-order optimality measure

$$\chi(x) \equiv \max_{\substack{x+w \in \Omega \\ \|w\| \le 1}} -\nabla f(x)^T w, \tag{3.4}$$

that has been used in previous analyses of GSS methods in the context of general linear constraints [18, 17, 20, 24]. This measure was introduced in [6, 5] and has the following three properties:

1. $\chi(x) \geq 0$,
2. $\chi(x)$ is continuous (if $\nabla f(x)$ is continuous), and
3. $\chi(x) = 0$ for $x \in \Omega$ if and only if x is a KKT point.

Thus any convergent sequence $\{x_k\}$ satisfying $\lim_{k\to\infty} \chi(x_k) = 0$ necessarily converges to a first-order stationary point.

3.2. Assumptions and conditions.

3.2.1. Conditions on the generating set. As in [20, 24], we require that $\kappa(\mathcal{G}_k)$, where $\mathcal{G}_k$ denotes the conforming directions generated in Steps 1, 13, and 22 of Algorithm 1, be uniformly bounded below.

Condition 3.2. There exists a constant $\kappa_{\min}$, independent of k, such that for every k for which $T(x_k, \epsilon_k) \neq \{0\}$, the set $\mathcal{G}_k$ generates $T(x_k, \epsilon_k)$ and satisfies $\kappa(\mathcal{G}_k) \geq \kappa_{\min}$, where $\kappa(\cdot)$ is defined in (3.2).

3.2.2. Conditions on the forcing function. Convergence theory for GSS methods typically requires either that all search directions lie on a rational lattice or that a sufficient decrease condition be imposed [18, 20]. This latter condition ensures that $f(x)$ is sufficiently reduced at each successful iteration. Both rational lattice and sufficient decrease conditions are mechanisms for globalization; each ensures that the step size ultimately becomes arbitrarily small if $f(x)$ is bounded below [18, 20, 17]. We only consider the sufficient decrease case because it allows a better choice of step; see [24, §8.2]. Specifically, we use the forcing function

$$\rho(\Delta) = \alpha\Delta^2,$$

where $\alpha > 0$ is specified by the user in Algorithm 3.

In general, the forcing function $\rho(\cdot)$ must satisfy Condition 3.3.

Condition 3.3. Requirements on the forcing function $\rho(\cdot)$:
1. $\rho(\cdot)$ is a nonnegative continuous function on $[0, +\infty)$.
2. $\rho(\cdot)$ is $o(t)$ as $t \downarrow 0$; i.e., $\lim_{t\downarrow 0} \rho(t) \,/\, t = 0$.
3. $\rho(\cdot)$ is nondecreasing; i.e., $\rho(t_1) \leq \rho(t_2)$ if $t_1 \leq t_2$.
4. $\rho(\cdot)$ is such that $\rho(t) > 0$ for $t > 0$.

Any forcing function may be substituted in Algorithm 3. For example, another valid forcing function is

$$\rho(\Delta) = \frac{\alpha\Delta^2}{\beta + \Delta^2}, \tag{3.5}$$

for $\alpha, \beta > 0$. The latter may offer some advantages because it does not require quadratic improvement for step sizes larger than 1 and consequently more trial points will be accepted as new minimums.

3.2.3. Assumptions on the objective function. We need to make some standard assumptions regarding the objective function. The first two assumptions do not require any continuity; only the third assumption requires that the gradient be Lipschitz continuous.

ASSUMPTION 3.4. The set $\mathcal{F} = \{ \, x \in \Omega \; | \; f(x) \leq f(x_0) \, \}$ is bounded.

ASSUMPTION 3.5. The function f is bounded below on Ω.

ASSUMPTION 3.6. The gradient of f is Lipschitz continuous with constant M on $\mathcal{F}$.

As in [20] we combine Assumptions 3.4 and 3.6 to assert the existence of a constant $\gamma > 0$ such that

$$\| \nabla f(x) \| \leq \gamma, \tag{3.6}$$

for all $x \in \mathcal{F}$.

3.2.4. Assumptions on the asynchronicity. In the synchronous case, we implicitly assume that the evaluation time for any single function evaluation is finite. However, in the asynchronous case, that assumption must be made explicit for the theory that follows even though it has no direct relevance to the algorithm. More discussion of this condition can be found in [17].

CONDITION 3.7. There exists an $\eta < +\infty$ such that the following holds. If a trial point is submitted to the evaluation queue at iteration k, either its evaluation will have been completed or it will have been pruned from the evaluation queue by iteration $k + \eta$.

3.3. Bounding a measure of stationarity. In this section, we prove global convergence for Algorithm 1 by showing (in Theorem 3.10) that $\chi(x_k)$ can be bounded in terms of the step size.

Synchronous GSS algorithms obtain optimality information at unsuccessful iterations because *all* points corresponding to the ϵ-tangent cone have been evaluated and rejected. In this case, we can bound $\chi(x)$ from (3.4) in terms of the step size Δ_k [20]. In asynchronous GSS, however, multiple unsuccessful iterations may pass before all points corresponding to generators of a specific ϵ-tangent cone have been evaluated. Proposition 3.8 says when we may be certain that all relevant points with respect to a specific ϵ-tangent cone have been evaluated and rejected.

PROPOSITION 3.8. *Suppose Algorithm 1 is applied to the optimization problem (1.1). Furthermore, at iteration k suppose we have*

$$\hat{\Delta}_k \equiv \max_{1 \leq i \leq p_k} \left\{ 2\Delta_k^{(i)} \right\} \leq \min(\Delta_{\min}, \epsilon_{\max}).$$

Let $\mathcal{G}$ be a set of generators for $T(x_k, \hat{\Delta}_k)$. Then $\mathcal{G} \subseteq \mathcal{D}_k$ and

$$f(x_k) - f(x_k + \hat{\Delta}_k d) \geq \rho(\hat{\Delta}_k) \textit{ for all } d \in \mathcal{G}. \tag{3.7}$$

Recall that $\rho(\cdot)$ is the forcing function discussed in §3.2.2.

Proof. Let $k^* \leq k$ be the most recent successful iteration (iteration zero is by default successful). Then $x_\ell = x_k$ for all $\ell \in \{k^*, \ldots, k\}$. Since $\hat{\Delta}_k \leq \Delta_{\min}$, there exists $\hat{k}$ with $k^* \leq \hat{k} \leq k$ such that $\delta_{\hat{k}} = \hat{\Delta}_k$ in either Step 12 or Step 21 of Algorithm 1. Moreover, since $\hat{\Delta}_k \leq \epsilon_{\max}$, we have $\epsilon_{\hat{k}} = \hat{\Delta}_k$ as well. Recalling that $\mathcal{G}$ comprises generators for $T(x_k, \hat{\Delta}_k) = T(x_{\hat{k}}, \epsilon_{\hat{k}})$, we have then that $\mathcal{G}$ was appended to $\mathcal{D}_{\hat{k}}$ (in either Step 14 or Step 23). Therefore, $\mathcal{G} \subseteq \mathcal{D}_k$ because there has been no successful iteration in the interim.

Now, every direction in $\mathcal{G}$ was appended with an initial step length greater than or equal to $\hat{\Delta}_k$. However, all the current step lengths are strictly less that $\hat{\Delta}_k$. Therefore, every point of the form

$$x_k + \hat{\Delta}_k d, \quad d \in \mathcal{G},$$

has been evaluated. (Note that, by definition of $T(x_k, \hat{\Delta}_k)$, $x_k + \hat{\Delta}_k d \in \Omega$ for all $d \in \mathcal{G}$. Hence $\tilde{\Delta}_k = \hat{\Delta}_k$ for all $d \in \mathcal{G}$.) None of these points has produced a successful iteration, and every one has parent x_k, therefore, (3.7) follows directly from Algorithm 3. ▯

Using the previous result, we can now show that the projection of the gradient onto a particular ϵ-tangent cone is bounded as a function of the step length Δ_k.

Theorem 3.9. *Consider the optimization problem (1.1), satisfying Assumption 3.6 along with Conditions 3.2 and 3.3. If*

$$\hat{\Delta}_k \equiv \max_{1 \leq i \leq p_k} \left\{ 2\Delta_k^{(i)} \right\} \leq \min(\Delta_{\min}, \epsilon_{\max}),$$

then

$$\| \, [-\nabla f(x_k)]_{T(x_k, \hat{\Delta}_k)} \, \| \leq \frac{1}{\kappa_{\min}} \left(M \hat{\Delta}_k + \frac{\rho(\hat{\Delta}_k)}{\hat{\Delta}_k} \right),$$

where the constant $\kappa_{\min}$ is from Condition 3.2 and M is the Lipschitz constant on $\nabla f(x)$ from Assumption 3.6.

Proof. Let $\mathcal{G}$ denote a set of generators for $T(x_k, \hat{\Delta}_k)$. By Condition 3.2 and (2.1), there exists a $\hat{d} \in \mathcal{G}$ such that

$$\kappa_{\min} \, \| \, [-\nabla f(x_k)]_{T(x_k, \hat{\Delta}_k)} \, \| \leq -\nabla f(x_k)^T \hat{d}. \tag{3.8}$$

Proposition 3.8 ensures that

$$f(x_k + \hat{\Delta}_k \hat{d}) - f(x_k) \geq -\rho(\hat{\Delta}_k).$$

The remainder of the proof follows [20, Theorem 6.3] and so is omitted. ▯

The previous result involves a specific ϵ-tangent cone. The next result generalizes this to bound the measure of stationarity $\chi(x_k)$ in terms of the step length Δ_k.

Theorem 3.10. *Suppose Assumptions 3.4 and 3.6 hold for (1.1) and that Algorithm 1 satisfies Conditions 3.2 and 3.3. Then if*

$$\hat{\Delta}_k \equiv \max_{1 \leq i \leq p_k} \left\{ 2\Delta_k^{(i)} \right\} \leq \min(\Delta_{\min}, \epsilon_{\max}),$$

we have

$$\chi(x_k) \leq \left(\frac{M}{\kappa_{\min}} + \frac{\gamma}{\nu_{\min}} \right) \hat{\Delta}_k + \frac{1}{\kappa_{\min}} \frac{\rho(\hat{\Delta}_k)}{\hat{\Delta}_k}. \tag{3.9}$$

Proof. This proof follows [20, Theorem 6.4] with appropriate modifications to use $\hat{\Delta}_k$ and so is omitted. ▯

3.4. Globalization. Next, in Theorem 3.12, we show that the maximum step size becomes arbitrarily close to zero. This is the globalization of GSS methods [18]. The proof hinges upon the following two properties of Algorithm 1 when Condition 3.7 holds:

1. The current smallest step length decreases by *at most* a factor of two at each unsuccessful iteration.
2. The current largest step-size decreases by *at least* a factor of two after every η consecutive unsuccessful iterations.

Before proving Theorem 3.12 we first prove the following proposition which says that, given any integer J, one can find a sequence of J or more consecutive unsuccessful iterations, i.e., the number of consecutive unsuccessful iterations necessarily becomes arbitrarily large.

Proposition 3.11. *Suppose that Assumption 3.5 holds for problem (1.1) and that Algorithm 1 satisfies Condition 3.3 and Condition 3.7. Let $\mathcal{S} = \{k_0, k_1, k_2, \dots\}$ denote the subsequence of successful iterations. If the number of successful iterations is infinite, then*

$$\limsup_{i\to\infty} (k_i - k_{i-1}) = \infty.$$

Proof. Suppose not. Then there exists an integer $J > 0$ such that $k_i - k_{i-1} < J$ for all i. We know that, at each unsuccessful iteration, the smallest step size either has no change or decreases by a factor of two. Furthermore, for any $k \in \mathcal{S}$, we have $\Delta_k^{(i)} \geq \Delta_{\min}$. Therefore, since a success must occur every J iterations, we have

$$\min_{1\leq i\leq |\mathcal{D}_k|} \left\{ \Delta_k^{(i)} \right\} \geq 2^{-J} \Delta_{\min}, \text{ for all } k.$$

Note the previous bound holds for all iterations, successful and unsuccessful.

Let $\hat{\mathcal{S}} = \{\ell_0, \ell_1, \ell_2, \dots\}$ denote an infinite subsequence of $\mathcal{S}$ with the additional property that its members are at least η apart, i.e.,

$$\ell_i - \ell_{i-1} \geq \eta.$$

Since the parent of any point x_k can be at most η iterations old by Condition 3.7, this sequence has the property that

$$f(x_{\ell_{i-1}}) \geq \textsc{ParentFx}(x_{\ell_i}) \text{ for all } i.$$

Combining the above with the fact that $\rho(\cdot)$ is nondecreasing from Condition 3.3, we have

$$f(x_{\ell_i}) - f(x_{\ell_{i-1}}) \leq f(x_{\ell_i}) - \textsc{ParentFx}(x_{\ell_i}) \leq -\rho(\hat{\Delta}) \leq -\rho(2^{-J}\Delta_{\min}) \equiv -\rho_\star,$$

where $\hat{\Delta} = \textsc{Step}(x_{\ell_i})$. Therefore,

$$\lim_{i\to\infty} f(x_{\ell_i}) - f(x_0) = \lim_{i\to\infty} \sum_{j=1}^{i} f(x_{\ell_j}) - f(x_{\ell_{j-1}}) \leq \lim_{i\to\infty} -i\rho_\star = -\infty,$$

contradicting Assumption 3.5. $\square$

THEOREM 3.12. *Suppose that Assumption 3.5 holds for problem (1.1) and that Algorithm 1 satisfies Condition 3.3 and Condition 3.7. Then*

$$\liminf_{k\to\infty} \max_{1\le i\le p_k} \left\{\Delta_k^{(i)}\right\} = 0.$$

Proof. Condition 3.7 implies that the current largest step-size decreases by *at least* a factor of two after every η consecutive unsuccessful iterations. Proposition 3.11 implies that the number of consecutive unsuccessful iterations can be made arbitrarily large. Thus the maximum step size can be made arbitrarily small and the result follows. □

3.5. Global convergence. Finally, we can combine Theorem 3.10 and Theorem 3.12 to immediately get our global convergence result.

THEOREM 3.13. *If problem (1.1) satisfies Assumptions 3.4, 3.5, and 3.6 and Algorithm 1 satisfies Conditions 3.2, 3.3, and 3.7, then*

$$\liminf_{k\to\infty} \chi(x_k) = 0.$$

4. Implementation Details. In this section we provide details of the implementation. For the most part we integrate the strategies outlined in [11, 14, 24].

4.1. Scaling. GSS methods are extremely sensitive to scaling, so it is important to use an appropriate scaling to get the best performance [11]. As is standard (c.f., [24]), we construct a positive, diagonal scaling matrix $S = \operatorname{diag}(s) \in \mathbb{R}^{n\times n}$ and a shift $r \in \mathbb{R}^n$ to define the transformed variables as

$$\hat{x} = S^{-1}(x - r).$$

Once we have computed an appropriate scaling matrix S and shift vector r, we transform (1.1) to

$$\begin{array}{ll} \text{minimize} & \hat{f}(\hat{x}) \\ \text{subject to} & \hat{c}_L \le \hat{A}_I \hat{x} \le \hat{c}_U \\ & \hat{A}_E \hat{x} = \hat{b}, \end{array} \tag{4.1}$$

where

$$\begin{aligned} \hat{f}(\hat{x}) &\equiv f(S\hat{x} + r) & \hat{A}_I &\equiv A_I S \\ \hat{A}_E &\equiv A_E S & \hat{c}_L &\equiv c_L - A_I r \\ \hat{b} &\equiv b - A_E r & \hat{c}_U &\equiv c_U - A_I r. \end{aligned}$$

Ideally, the simple bounds are transformed to the unit hypercube:

$$\{\, \hat{x} \mid 0 \le \hat{x} \le e \,\}.$$

In the numerical experiments in §5, we used

$$s_i = \begin{cases} u_i - \ell_i & \text{if } u_i,\, \ell_i \text{ are finite} \\ 1 & \text{otherwise,} \end{cases} \quad \text{and } r_i = \begin{cases} \ell_i & \text{if } \ell_i > -\infty \\ 0 & \text{otherwise.} \end{cases}$$

From this point forward, we use the notation in (1.1) but assume that the problem is appropriately scaled, i.e., as in (4.1).

The theory is not affected by scaling the variables, but differently scaled variables tend to make GSS methods very slow. For instance, Hough et al. [15] study a circuit simulation problem where the variable ranges differ by more than 10 orders of magnitude, but APPS is able to solve an appropriately scaled version of the problem.

4.2. Function value caching. In the context of generating set search algorithms, we frequently re-encounter the same trial points. In order to avoid repeating expensive function evaluations, we cache the function value of every point that is evaluated. Moreover, cached values can be used across multiple optimization runs.

An important feature of our implementation is that we do not require that points be exactly equal in order to use the cache. Instead, we say that two points, x and y, are ξ-equal if

$$|y_i - x_i| \leq \xi \; s_i, \text{ for } i = 1, 2, \ldots, n.$$

Here ξ is the *cache comparison tolerance* and s_i is the scaling of the ith variable. Note that this comparison function corresponds to a lexicographic ordering of the points. Consequently, we can store them in a binary splay tree which in turn provides extremely efficient look-ups [14].

If the two-norm based comparison

$$\| \, y - x \, \| \leq \xi,$$

is used as in [24], then the work to do a cache look-up grows linearly with the number of cached points and is inefficient.

The choice of ξ should reflect the sensitivity of the objective function. In our experience, practitioners often have a sense what this should be. If a 1% change in the variables is not expected to impact the objective function, than choosing $\xi = 0.01$ is clearly reasonable. Setting $\xi = 0$ forces an exact match (within machine precision) and thus conforms with the convergence theory. The default $\xi = .5\Delta_{\text{tol}}$ is half as big as the smallest step size that the algorithm can take (unless the boundary is nearby). This means that the stopping condition is ideally using truly distinct function evaluations to make its stopping decision. The significant reduction in function evaluations is the reward for relaxing the comparison tolerance (see §5).

4.3. Snapping to the boundary. In Algorithm 2, we modify the step length in order to step exactly to the boundary whenever the full step would have produced an infeasible trial point. Conversely, it is sometimes useful to "snap" feasible trial points to the boundary when they are very close to it because, in real-world applications, it is not uncommon for the objective function to be highly sensitive to whether or not a constraint is active. For example, an "on/off" switch may be activated in the underlying simulation only if a given x_i lies on its bound. A further subtle point is that if a function value cache like that in §4.2 is used, the cache lookup may preclude evaluation of points on the boundary that lie within the cache tolerance neighborhood of a previously evaluated point that is not on the boundary. This modification is wholly based on practical experience and not justified by the theory, yet is has not negatively affected convergence (using the default value of the cache tolerance parameter) in our experience.

Suppose that x is a trial point produced by Algorithm 2. We modify the point x as follows. Let $\mathcal{S}$ denote the set of constraints within a distance ϵ_{snap} of x, defining the system

$$(A_I)_{\mathcal{S}} z = (c_I)_{\mathcal{S}}, \tag{4.2}$$

where (c_I) represents the appropriate lower or upper bound, whichever is active. We prune dependent rows from (4.2) so that the matrix has full row rank. A row is considered dependent if the corresponding diagonal entry for the matrix R from the QR factorization of A^T is less than 10^{-12}. LAPACK is then used to find the point z that minimizes $\| x - z \|$ subject to the equality constraints (4.2). If the solution z to the generalized least-squares problem is feasible for (1.1), reset $x = z$ before sending the trial point to the evaluation queue. Note that this modification is included for practical concerns mentioned in the preceding paragraph and is not a theoretically necessary modification to x; hence, if the LAPACK fails to find a solution to the generalized least-squares problem due to near linear dependency in the constraint matrix, we simply use the original point x. Like the caching of function values, the proposed modification is based upon the final step tolerance chosen by the user; which as stated in Theorem 3.10, denotes an implicit bound on the KKT conditions. Thus the modifications are on the order of the accuracy specified by the user.

4.4. Generating conforming search directions. In Steps 1, 13, and 22, we need to compute generators for the tangent cones corresponding to ϵ-active constraints. In the unconstrained and bound-constrained cases, the $2n$ coordinate directions always include an appropriate set of generators. For linear constraints, however, this is not the case; instead, the set of directions depends on A_I and A_E.

Our method for generating appropriate conforming search directions follows [24]. Let V_P and V_L be formed as in (3.1). Whenever possible the following theorem is used to compute a set of generators for $T(x, \epsilon)$.

THEOREM 4.1 ([24]). *Suppose $N(x, \epsilon)$ is generated by the positive span of the columns of the matrix V_P and the linear span of the columns of the matrix V_L :*

$$N(x, \epsilon) = \{ v \mid v = V_P \lambda + V_L \alpha, \ \lambda \geq 0 \},$$

where λ and α denote vectors of appropriate length. Let Z be a matrix whose columns are a basis for the nullspace of V_L^T, and N be a matrix whose columns are a basis for the nullspace of $V_P^T Z$. Finally, suppose $V_P^T Z$ has full row rank implying a matrix R exists satisfying $V_P^T Z R = I$. Then $T(x, \epsilon)$ is the positive span of the columns of $-ZR$ together with the linear span of the columns of ZN :

$$T(x, \epsilon) = \{ w \mid w = -ZRu + ZN\xi, \ u \geq 0 \},$$

where u and ξ denote vectors of appropriate length.

In order to determine if Theorem 4.1 is applicable, LAPACK [1] is used to compute a singular value decomposition of $V_P^T Z$. If the number of singular values greater than 10^{-12} equals the number of rows in $V_P^T Z$, we say $V_P^T Z$ has full row rank. We then construct R from the singular value decomposition of $V_P^T Z$. Thus, whenever possible, linear algebra is used to explicitly compute generators for $T(x, \epsilon)$. The following corollary provides an upper bound on the number of directions necessary in this case.

COROLLARY 4.2. *Suppose that generators $\mathcal{G}_k$ for the tangent cone $T(x, \epsilon)$ are computed according to Theorem 4.1. Then*

$$|\mathcal{G}_k| = 2n_z - n_r,$$

where $n_z = \dim(Z)$ *and* $n_r = \dim(V_P^T Z)$. *In particular,*

$$|\mathcal{G}_k| \leq 2n.$$

Proof. We know that the magnitude of $\mathcal{G}_k$ is given by the number of columns in R plus *twice* the number of columns in N. Since R denotes the pseudoinverse of $V_P^T Z$ and N its nullspace basis matrix we must have that R is an $n_z \times n_r$ matrix and N an $n_z \times (n_z - r)$ matrix where

$$n_z = \dim(Z) \quad \text{and} \quad n_r = \dim(V_P^T Z).$$

Thus the total number of generators is given by

$$n_r + 2(n_z - n_r) = 2n_z - n_r \leq 2n,$$

since $n_z \leq n$. ◻

Note that whenever Theorem 4.1 is applicable, Corollary 4.2 implies that the number of search directions is less than twice the dimension of the nullspace of the equality constraints; furthermore, the presence of inequality constraints can only reduce this quantity.

If we are unable to apply Theorem 4.1, i.e., if $V_P^T Z$ fails to have a right inverse, we use the C-library cddlib developed by Komei Fukuda [8] that implements the double description method of Motzkin et al. [29]. In this case there is no upper bound on the number of necessary search directions in terms of the number of variables. In fact, though the total number of necessary search directions is always finite (see [20]), their number can be combinatorially large. In §5 we report a case where the ϵ-active constraints encountered require more than 2^{20} vectors to generate the corresponding tangent cone. Fortunately, this combinatorial explosion appears to be more a worst-case example than something we would expect to encounter in real life; in all remaining results presented in §5, a modest number of search directions was required when Theorem 4.1 was inapplicable.

4.5. Direction caching. Further efficiency can be achieved through the caching of tangent cone generators. Every time a new set of generators is computed, it can be cached according to the set of active constraints. Moreover, even when ϵ_k changes, it is important to track whether or not the set of active constraints actually changes as well. Results on using cached directions are reported in §5. In problem EXPFITC, the search directions are modified to incorporate new ϵ-active constraints 98 times. However, because generators are cached, new directions are only computed 58 times and the cache is used 40 times.

In order for Condition 3.2 to be satisfied, $\bigcup_{k=1}^{+\infty} D_k$ should be finite [20]. Reusing the same generators every time the same active constraints are encountered ensures this is the case in theory and practice.

4.6. Augmenting the search directions. The purpose of forming generators for $T(x_k, \epsilon_k)$ is to allow tangential movement along nearby constraints ensuring that the locally feasible region is sufficiently explored. But these directions necessarily do not point towards the boundary. In order to allow boundary points to be approached directly, additional search directions can be added. Two candidates for extra search directions are shown in Figure 4.1. In our experiments the (projected) constraint normals were added to the corresponding set of conforming search directions. That

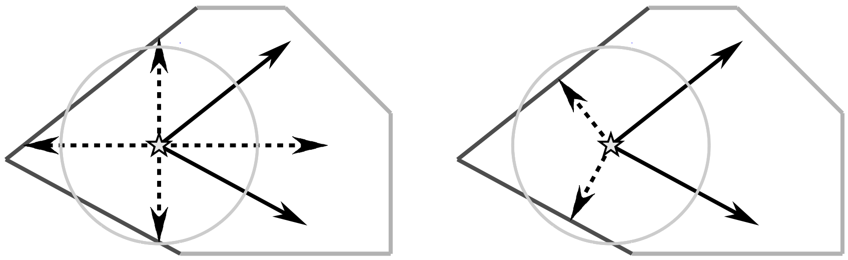


FIG. 4.1. *Two options for additional search directions are the coordinate directions (left) or the normals to the linear inequality constraints (right).*

is, we append the columns of the matrix $(ZZ)^T V_P$, where Z and V_P are defined in Theorem 4.1.

Augmenting the search directions is allowed by the theory and tends to reduce the overall run time because it enables direct steps to the boundary.

5. Numerical results. Our goal is to numerically verify the effectiveness of the asynchronous GSS algorithm for linearly-constrained problems. Algorithm 1 is implemented in APPSPACK Version 5.0.1, including all the implementation enhancements outlined in §4. All problems were tested on Sandia's Institutional Computing Cluster (ICC) with 3.06GHz Xeon processors and 2GB RAM per node.

5.1. Test Problems. We test our method on problems from the CUTEr (Constrained and Unconstrained Testing Environment, revisited) test set [5]. We selected every problem with general linear constraints and 1000 or fewer variables, for a total of 119 problems. We divide these problems into three groups:

- Small (1–10 variables): 72 *(6 of which have trivial feasible regions)*
- Medium (11–100 variables): 24
- Large (101–1000 variables): 23

The CUTEr test set is specifically designed to challenge even the most robust, derivative-based optimization codes. Consequently, we do not expect to be able to solve all of the test problems. Instead, our goal is to demonstrate that we can solve more problems than have ever been solved before using a derivative-free approach, including problems with constraint degeneracies. To the best of our knowledge, this is the largest set of test problems ever attempted with a derivative-free method for linearly-constrained optimization.

5.2. Choosing a starting point. In general, we used the initial points provided by CUTEr. If the provided point was infeasible, however, we instead found a starting point by solving the following program using MATLAB's `linprog` function:

$$\begin{array}{ll} \text{minimize} & 0 \\ \text{subject to} & \begin{array}{ccccc} c_L & \leq & A_I x & \leq & c_U \\ & & A_E x & = & b. \end{array} \end{array} \tag{5.1}$$

If the computed solution to the first problem was still infeasible, we applied MATLAB's `quadprog` function to

$$
\begin{array}{lrcccl}
\text{minimize} & & & \| x - x_0 \|_2^2 & & \\
\text{subject to} & c_L & \leq & A_I x & \leq & c_U \\
 & & & A_E x & = & b.
\end{array}
\tag{5.2}
$$

Here, x_0 is the (infeasible) initial point provided by CUTEr. Using this approach, we were able to find feasible starting points for every problem save ACG, HIMMELBJ, and NASH.

5.3. Parameter Choices. The following parameters were used to initialize Algorithm 1: (a) $\Delta_{\text{tol}} = 1.0 \times 10^{-5}$, (b) $\Delta_{\min} = 2.0 \times 10^{-5}$, (c) $\delta_0 = 1$, (d) $\epsilon_{\max} = 2.0 \times 10^{-5}$, (e) $q_{\max}$ = number of processors, and (f) $\alpha = 0.01$. Additionally, for the snap procedure outlined in §4.3, we used $\epsilon_{\text{snap}} = 0.5 \times 10^{-5}$. We limited the number of function evaluations to 10^6 and put a lower bound on the objective value of -10^9 (as a limit for unboundedness). For extra search directions, as described in §4.6, we added the outward pointing constraint normals.

5.4. Numerical results. Numerical results on all the test problems are presented in Tables 5.1–5.4. Detailed descriptions of what each column indicates are shown in Figure 5.1. Note that the sum of F-Evals and F-Cached yields the total number of function evaluations; likewise, the sum of D-LAPACK, D-CDDLIB, and D-Cached is the number of times that directions needed to be computed because the set of ϵ-active constraints changed.

- **Problem:** Name of the CUTEr test problem.
- $\mathbf{n/m_b/m_i/m_e}$**:** Number of variables, bound constraints, inequality constraints, and equality constraints, respectively.
- $\mathbf{f(x^*)}$**:** Final solution.
- **Soln. Acc.:** Relative accuracy of solution as compared to SNOPT [9]:

$$Re(\alpha, \beta) = \frac{\beta - \alpha}{\max\{1, |\alpha|, |\beta|)\}},$$

where α is the final APPSPACK objective value and β is the final SNOPT objective value. A positive value indicates that the APPSPACK solution is better than SNOPT's.
- **F-Evals:** Number of *actual* function evaluations, i.e., not counting cached function values.
- **F-Cached:** Number of times that cached function values were used.
- **Time (sec):** Total parallel run-time.
- **D-LAPACK/D-CDDLIB:** Number of times that LAPACK or CDDLIB was called, respectively, to compute the search directions.
- **D-Cached:** Number of times that a cached set of search directions was used.
- **D-MaxSize:** Maximum number of search directions ever used for a single iteration.
- **D-Appends:** Number of times that additional search directions had to be appended in Step 23.

Fig. 5.1. *Column descriptions for numerical results.*

Because each run of an asynchronous algorithm can be different, we ran each problem a total of ten times and present averaged results. The exception is the objective value $f(x^*)$, for which we present the best solution. Problems which had multiple local minima (i.e., whose relative difference between best and worst objective value is greater than 10^{-5}) are denoted in the tables by an asterisk and Table 5.5 explicitly gives the differences for those cases.

Problem	$n/m_b/m_e/\ m_i$	$f(x^*)$	Soln. Acc.	F-Evals	F-Cached	Time(sec)	D-LAPACK	D-CDDLIB	D-Cached	D-MaxSize	D-Appends
AVGASA	8/ 16/ 0/ 10	-4.6	-6e-11	490/	50	1.4	6/	0/	0	16	0
AVGASB	8/ 16/ 0/ 10	-4.5	2e-11	526/	56	1.2	9/	0/	0	16	0
BIGGSC4	4/ 8/ 0/ 13	-2.4e1	0	108/	11	1.3	4/	0/	0	8	0
BOOTH	2/ 0/ 2/ 0	*N/A — equality constraints determine solution*									
BT3	5/ 0/ 3/ 0	4.1	-5e-11	127/	35	1.2	1/	0/	0	4	0
DUALC1	9/ 18/ 1/214	6.2e3	-1e-11	866/	140	1.5	10/	0/	0	16	0
DUALC2	7/ 14/ 1/228	3.6e3	6e-12	278/	35	1.2	10/	0/	0	12	0
DUALC5	8/ 16/ 1/277	4.3e2	-3e-10	527/	58	1.5	6/	1/	0	15	0
DUALC8	8/ 16/ 1/502	1.8e4	-6e-10	503/	58	1.4	9/	0/	4	14	0
EQC	9/ 0/ 0/ 3	*N/A — upper bound less than lower bound*									
EXPFITA	5/ 0/ 0/ 22	1.1e-3	-3e-10	1081/	501	1.5	11/	0/	0	10	0
EXPFITB	5/ 0/ 0/102	5.0e-3	-5e-10	467/	175	1.2	29/	0/	0	10	0
EXPFITC	5/ 0/ 0/502	2.3e-2	-2e-8	3372/	1471	3.4	17/	41/	18	32	11
EXTRASIM	2/ 1/ 1/ 0	1.0	0	18/	1	1.0	2/	0/	1	2	0
GENHS28	10/ 0/ 8/ 0	9.3e-1	-2e-10	143/	45	1.1	1/	0/	0	4	0
HATFLDH	4/ 8/ 0/ 13	-2.4e1	0	121/	12	1.2	4/	0/	0	8	0
HIMMELBA	2/ 0/ 2/ 0	*N/A — upper bound less than lower bound*									
HONG	4/ 8/ 1/ 0	2.3e1	-4e-11	245/	53	1.0	4/	0/	2	6	0
HS105	8/ 16/ 0/ 1	1.0e3	-1e-11	1447/	192	1.4	5/	0/	0	16	0
HS112	10/ 10/ 3/ 0	-4.8e1	-3e-9	1810/	166	1.2	13/	0/	1	14	0
HS21	2/ 4/ 0/ 1	-1.0e2	-8e-10	88/	21	1.1	1/	0/	0	4	0
HS21MOD	7/ 8/ 0/ 1	-9.6e1	-1e-16	1506/	254	1.2	3/	0/	2	14	0
HS24	2/ 2/ 0/ 3	-1.0	-4e-10	67/	6	1.1	2/	0/	1	4	0
HS268	5/ 0/ 0/ 5	*Failed — evaluations exhausted*									
HS28	3/ 0/ 1/ 0	0.0	0	145/	52	1.4	1/	0/	0	4	0
HS35	3/ 3/ 0/ 1	1.1e-1	1e-10	171/	33	1.0	1/	0/	0	6	0
HS35I	3/ 6/ 0/ 1	1.1e-1	-9e-10	124/	30	1.0	1/	0/	0	6	0
HS35MOD	3/ 4/ 0/ 1	2.5e-1	0	73/	1	1.1	2/	0/	0	4	0
HS36	3/ 6/ 0/ 1	-3.3e3	0	81/	3	1.1	3/	0/	0	6	0
HS37	3/ 6/ 0/ 2	-3.5e3	-8e-11	131/	25	1.1	1/	0/	0	6	0
HS41	4/ 8/ 1/ 0	1.9	-5e-11	168/	35	1.0	2/	0/	0	6	0
HS44*	4/ 4/ 0/ 6	-1.5e1	0	99/	10	1.0	5/	0/	0	8	0
HS44NEW *	4/ 4/ 0/ 6	-1.5e1	0	137/	12	1.1	4/	0/	0	8	0
HS48	5/ 0/ 2/ 0	0.0	0	261/	69	1.8	1/	0/	0	6	0
HS49	5/ 0/ 2/ 0	1.6e-7	-2e-7	24525/	8315	3.2	1/	0/	0	6	0
HS50	5/ 0/ 3/ 0	0.0	0	279/	99	1.5	1/	0/	0	4	0

TABLE 5.1A

CUTEr problems with 10 or fewer variables, tested on 20 processors.

5.4.1. Group 1: 1–10 Variables. Consider first Tables 5.1a and 5.1b, which show results for 72 linearly-constrained CUTEr problems with up to 10 variables. Note that some of the problems had as many as 2000 inequality constraints. Six of the problems had non-existent or trivial feasible regions and so are excluded from our analysis. Of the 66 remaining problems, APPSPACK was able to solve 63 (95%).

The final objective function obtained by APPSPACK compared favorably to that obtained by SNOPT, a derivative-based code. We compare against SNOPT to illus-

Problem	$n/m_b/m_e/\ m_i$	$f(x^*)$	Soln. Acc.	F-Evals/ F-Cached	Time(sec)	D-LAPACK/ D-CDDLIB/ D-Cached	D-MaxSize	D-Appends
HS51	5/ 0/ 3/ 0	0.0	0	110/ 34	1.0	1/ 0/ 0	4	0
HS52	5/ 0/ 3/ 0	5.3	-9e-11	123/ 42	1.0	1/ 0/ 0	4	0
HS53	5/ 10/ 3/ 0	4.1	-1e-9	122/ 41	2.1	2/ 0/ 1	4	0
HS54	6/ 12/ 1/ 0	-1.9e-1	2e-1	271/ 42	1.2	3/ 0/ 1	10	0
HS55	6/ 8/ 6/ 0	6.3	5e-11	20/ 1	1.1	1/ 1/ 0	3	0
HS62	3/ 6/ 1/ 0	-2.6e4	-6e-10	376/ 146	1.1	1/ 0/ 0	4	0
HS76	4/ 4/ 0/ 3	-4.7	4e-11	243/ 39	1.1	3/ 0/ 0	8	0
HS76I	4/ 8/ 0/ 3	-4.7	4e-11	208/ 36	1.8	3/ 0/ 0	8	0
HS86	5/ 5/ 0/ 10	-3.2e1	-3e-11	160/ 29	1.2	4/ 1/ 0	11	0
HS9	2/ 0/ 1/ 0	-5.0e-1	0	57/ 5	1.1	1/ 0/ 0	2	0
HUBFIT	2/ 1/ 0/ 1	1.7e-2	-1e-10	68/ 18	1.4	2/ 0/ 0	4	0
LIN	4/ 8/ 2/ 0	-1.8e-2	-2e-10	68/ 0	1.2	1/ 0/ 0	4	0
LSQFIT	2/ 1/ 0/ 1	3.4e-2	-1e-10	67/ 20	1.1	2/ 0/ 0	4	0
ODFITS	10/ 10/ 6/ 0	-2.4e3	1e-12	15807/4695	3.0	1/ 0/ 0	8	0
OET1	3/ 0/ 0/1002	5.4e-1	3e-10	722/ 148	2.1	0/ 8/ 0	8	0
OET3	4/ 0/ 0/1002	4.5e-3	-6e-7	1347/ 344	2.5	2/ 8/ 0	107	1
PENTAGON	6/ 0/ 0/ 15	1.4e-4	-3e-10	3089/ 783	1.5	4/ 0/ 0	12	0
PT	2/ 0/ 0/ 501	1.8e-1	4e-10	496/ 159	1.6	0/ 17/ 0	10	0
QC	9/ 18/ 0/ 4	-9.6e2	4e-12	188/ 9	1.6	8/ 0/ 0	14	0
QCNEW	9/ 0/ 0/ 3	*N/A — upper bound less than lower bound*						
S268	5/ 0/ 0/ 5	*Failed — evaluations exhausted*						
SIMPLLPA	2/ 2/ 0/ 2	1.0	0	452/ 110	1.1	2/ 0/ 0	4	0
SIMPLLPB	2/ 2/ 0/ 3	1.1	0	382/ 97	1.3	3/ 0/ 0	4	0
SIPOW1	2/ 0/ 0/2000	-1.0	0	130/ 233	2.0	0/104/ 1	6	0
SIPOW1M	2/ 0/ 0/2000	-1.0	0	137/ 227	2.0	0/100/ 0	6	0
SIPOW2	2/ 0/ 0/2000	-1.0	0	176/ 324	1.9	148/ 0/ 0	4	0
SIPOW2M	2/ 0/ 0/2000	-1.0	0	179/ 324	2.2	149/ 0/ 0	4	0
SIPOW3	4/ 0/ 0/2000	5.3e-1	-1e-10	1139/ 252	3.7	115/ 4/ 0	13	1
SIPOW4	4/ 0/ 0/2000	*Failed — empty tangent cone encountered*						
STANCMIN	3/ 3/ 0/ 2	4.2	0	69/ 21	1.0	3/ 0/ 0	6	0
SUPERSIM	2/ 1/ 2/ 0	*N/A — equality constraints determine solution*						
TAME	2/ 2/ 1/ 0	0.0	0	38/ 22	1.6	2/ 0/ 0	2	0
TFI2	3/ 0/ 0/ 101	6.5e-1	0	695/ 175	1.2	36/ 0/ 0	6	0
TFI3	3/ 0/ 0/ 101	4.3	7e-11	83/ 31	1.1	13/ 0/ 0	6	0
ZANGWIL3	3/ 0/ 3/ 0	*N/A — equality constraints determine solution*						
ZECEVIC2	2/ 4/ 0/ 2	-4.1	-7e-10	66/ 30	1.1	1/ 0/ 0	4	0

TABLE 5.1B

CUTEr problems with 10 or fewer variables, tested on 20 processors.

trate that it is possible to obtain the same objective values. In general, if derivatives are readily available, using a derivative-based code such as SNOPT is preferred. We do note, however, that APPSPACK converged to different solutions on different runs on HS44 and HS44NEW. This is possibly due to the problems having multiple local minima. Otherwise, APPSPACK did at least as well as SNOPT on all 63 problems, comparing six digits of relative accuracy. In fact, the difference between objective values was greater than 10^{-6} on only one problem, HS54. In this case APPSPACK converged to a value of `-.19` while SNOPT converged to 0. Again, we attribute such differences to these problems having multiple local minima.

In a few cases, the number of function evaluations (F-Evals) is exceedingly high (e.g., `HS49` or `ODFITS`). This is partly due to the tight stopping tolerance ($\Delta_{\max} = 10^{-5}$). In practice, we typically recommend a stop tolerance of $\Delta_{\max} = 10^{-2}$. GSS methods, like steepest descent, quickly find the neighborhood of the solution but are slow to converge to the exact minimum. An example of this behavior is provided, for example, in [24].

In general, the set of search directions changed many times over the course of the iterations. The sum of D-LAPACK and D-CDDLIB is the total number of times an entirely new set of ϵ-active constraints was encountered. The value of D-Cached is the number of times that a previously encountered set of ϵ-active constraints is encountered again. In general, a new set of ϵ-active constraints will yield a different set of search directions. In a few cases, only one set of search directions was needed to solve the entire problem (cf., `HS28/35/37`, etc.), which can be due to having a small number of constraints or only equality constraints. In other cases, a large number of different sets of search direction was needed (cf., `SIPOW2/2M/3`). It is important to have the capability to handle degenerate vertices; 12 (19%) of the problems that were solved required CDDLIB to generate search directions.

The total number of search directions required at any single iteration (D-MaxSize) was $2n$ or less in 55 (87%) of the cases. The number of search directions can be larger than $2n$ if constraint degeneracy is encountered and/or additional search directions are appended in Step 23. Problem `OET3` required 107 at a single iteration. The need to append search directions (D-Appends), which is unique to the asynchronous method, occurred in 3 (4%) cases. We attribute this to the benefits of choosing a small value of $\epsilon_{\max}$.

5.4.2. Group 2: 11–100 Variables. Of the 24 problems in this category, we were unable to identify feasible starting points in 2 cases, so we ignore these for our analyses. We were able to solve 16 (73%) of the remaining 22 problems. The problem of encountering an empty tangent cone, which happened in 3 cases, is like the situation shown in Figure 2.1(d). It can happen as a function of poor scaling of the variables when $\epsilon_{\max}$ is too large. Problem `MAKELA` is famously degenerate and requires $2^{20} + 1$ generators [24]. On only one problem, `KSIP`, was the difference in solutions between APPSPACK and SNOPT greater than 10^{-6}.

Five problems (31%) require more than 50,000 function evaluations. We can only hope that such behavior does not typify real-world problems with expensive evaluations. As noted previously, the number of evaluations will be greatly reduced if Δ_{tol} is increased.

The number of search directions exceeded $2n$ for five problems. The problem `KSIP` required 4144 search directions at one iteration. The problem `DUAL1` required 137 appends to the search directions.

We have selected a subset of these moderate-sized problems to compare the synchronous and asynchronous approaches. A major motivation for the asynchronous version of GSS is to reduce overall parallel runtime. In our experience, many real-world problems have function evaluation times that are measured in minutes or hours and that vary substantially from run to run. In these cases, load balancing is a major issue. In these comparisons, the synchronous is the same algorithm (and software) except that all trial point evaluations must finish before the algorithm can proceed

Problem	$n/\ m_b/m_e/\ \ m_i$	$f(x^*)$	Soln. Acc.	F-Evals / F-Cached	Time(sec)	D-LAPACK / D-CDDLIB / D-Cached	D-MaxSize	D-Appends
AVION2	49/ 98/ 15/ 0	*Failed — evaluations exhausted*						
DEGENLPA	20/ 40/ 15/ 0	*Failed — empty tangent cone encountered*						
DEGENLPB	20/ 40/ 15/ 0	*Failed — empty tangent cone encountered*						
DUAL1	85/170/ 1/ 0	3.5e-2	-1e-7	469011/2893	251.2	142/1/456	301	137
DUAL2	96/192/ 1/ 0	3.4e-2	-4e-8	179609/ 973	121.6	150/1/ 23	191	0
DUAL4	75/150/ 1/ 0	7.5e-1	-3e-8	56124/3178	29.3	90/1/ 13	283	1
FCCU	19/ 19/ 8/ 0	1.1e1	-9e-11	4461/ 358	1.6	7/2/ 1	23	0
GOFFIN	51/ 0/ 0/ 50	0.0	0	13728/ 488	4.3	1/0/ 0	102	0
HIMMELBI	100/200/ 0/ 12	-1.7e3	-6e-10	142476/2720	90.3	94/0/ 18	235	4
HIMMELBJ	45/ 0/ 14/ 0	*N/A — could not find initial feasible point*						
HS118	15/ 30/ 0/ 29	6.6e2	-2e-16	635/ 72	1.2	21/0/ 0	32	0
HS119	16/ 32/ 8/ 0	2.4e2	-3e-11	479/ 34	1.2	14/0/ 0	16	0
KSIP	20/ 0/ 0/1001	1.0	-3e-1	3107/ 120	136.3	2/4/ 0	4144	0
LOADBAL	31/ 42/ 11/ 20	4.5e-1	4e-9	51262/2909	9.1	11/0/ 0	40	0
LOTSCHD	12/ 12/ 7/ 0	2.4e3	-1e-11	270/ 28	1.4	6/0/ 0	10	0
MAKELA	21/ 0/ 0/ 40	*Failed — too many generators*						
NASH	72/ 0/ 24/ 0	*N/A — could not find initial feasible point*						
PORTFL1	12/ 24/ 1/ 0	2.0e-2	-3e-10	946/ 104	1.4	9/0/ 2	22	0
PORTFL2	12/ 24/ 1/ 0	3.0e-2	1e-9	919/ 106	1.3	8/0/ 0	22	0
PORTFL3	12/ 24/ 1/ 0	3.3e-2	4e-10	997/ 109	1.4	10/0/ 0	22	0
PORTFL4	12/ 24/ 1/ 0	2.6e-2	-1e-10	917/ 96	1.2	8/0/ 0	22	0
PORTFL6	12/ 24/ 1/ 0	2.6e-2	4e-9	1098/ 103	1.4	8/0/ 0	22	0
QPCBLEND	83/ 83/ 43/ 31	*Failed — empty tangent cone encountered*						
QPNBLEND	83/ 83/ 43/ 31	*Failed — evaluations exhausted*						

Table 5.2
CUTEr problems with 11–100 variables, tested on 40 processors.

beyond Step 8, i.e., $\mathcal{Y}_k = X_k$. This comparison may not be ideal but, to the best of our knowledge, there are no other synchronous parallel versions of pattern search or GSS to which to compare.

Ideally, we would compare these methods on a collection of real-world problems, but no such test set exists. We have used this method to solve real-world problems, but none that is appropriate for publication. Thus, we compare these methods on a subset of the CUTEr test problems. To make this more like real-world problems, we add a random time delay of 5 to 15 seconds for each evaluation which corresponds to a difference of three times between the slowest and fastest evaluations; this is very realistic in our experience. An advantage of this approach is that it is reproducible. We ran each problem on 5, 10, and 20 processors. Table 5.3 shows the time and number of function evaluation for each problem; Figure 5.2 show the results as a bar graph.

Although the asynchronous code did more function evaluations overall, the total time to solution was reduced in every case save two (PORTFL4 on 5 processors and LOTSCHD on 20 processors). Thus the asynchronous approach not only gained more information in less time, but solved each problem in less time. This suggests that

Problem	$n/m_b/m_e/m_i$	Sync/Async	Processors	F-Evals F-Cached	Time (sec)	D-LAPACK D-CDDLIB D-Cached	D-MaxSize	D-Appends
FCCU	19/ 19/ 8/ 0	S	5	4444/348	15361.9	7/2/ 1	23	0
		A	5	3689/216	11649.1	17/1/ 9	23	0
		S	10	4446/347	7788.6	7/2/ 1	23	0
		A	10	4166/173	5817.7	26/1/78	23	0
		S	20	4444/349	4686.2	7/2/ 1	23	0
		A	20	5133/239	3513.2	15/2/64	23	0
HS118	15/ 30/ 0/ 29	S	5	624/ 54	2259.2	21/0/ 0	30	0
		A	5	553/ 93	1815.3	40/0/ 4	30	0
		S	10	624/ 54	1168.2	21/0/ 0	30	0
		A	10	648/ 94	939.9	47/0/ 2	30	0
		S	20	624/ 54	772.8	21/0/ 0	30	0
		A	20	829/225	667.4	48/0/ 1	30	0
HS119	16/ 32/ 8/ 0	S	5	471/ 34	1839.7	13/0/ 0	16	0
		A	5	481/ 42	1594.9	18/0/ 0	16	0
		S	10	464/ 35	1045.4	13/0/ 0	16	0
		A	10	545/ 46	801.1	20/0/ 1	16	0
		S	20	474/ 36	822.4	13/0/ 0	16	0
		A	20	638/ 56	586.6	21/0/ 1	16	0
LOTSCHD	12/ 12/ 7/ 0	S	5	267/ 26	1148.8	6/0/ 0	10	0
		A	5	339/ 38	1112.9	6/0/ 9	10	0
		S	10	267/ 26	696.8	6/0/ 0	10	0
		A	10	398/ 44	722.6	7/0/14	10	0
		S	20	267/ 26	607.9	6/0/ 0	10	0
		A	20	464/ 49	585.8	7/0/12	10	0
PORTFL1	12/ 24/ 1/ 0	S	5	918/111	3351.2	9/0/ 2	22	0
		A	5	973/ 91	3166.6	10/0/ 1	22	0
		S	10	918/111	1818.8	9/0/ 2	22	0
		A	10	1155/106	1696.9	9/0/ 2	22	0
		S	20	918/111	1161.7	9/0/ 2	22	0
		A	20	1422/107	1033.8	11/0/ 4	22	0
PORTFL2	12/ 24/ 1/ 0	S	5	963/111	3513.4	8/0/ 0	22	0
		A	5	808/ 90	2634.7	6/0/ 0	22	0
		S	10	961/113	1912.0	8/0/ 0	22	0
		A	10	980/ 84	1455.6	6/0/ 0	22	0
		S	20	962/112	1261.3	8/0/ 0	22	0
		A	20	1258/ 92	930.5	9/0/ 2	22	0
PORTFL3	12/ 24/ 1/ 0	S	5	975/109	3544.1	11/0/ 0	22	0
		A	5	771/ 80	2510.8	7/0/ 0	22	0
		S	10	973/111	1911.8	11/0/ 0	22	0
		A	10	973/ 92	1442.3	8/0/ 2	22	0
		S	20	971/113	1210.7	11/0/ 0	22	0
		A	20	1376/102	998.7	13/0/ 5	22	0
PORTFL4	12/ 24/ 1/ 0	S	5	874/ 94	3157.9	7/0/ 0	22	0
		A	5	1148/110	3714.8	11/0/ 4	22	0
		S	10	872/ 96	1722.7	7/0/ 0	22	0
		A	10	1157/ 94	1706.8	11/0/ 6	22	0
		S	20	873/ 95	1061.2	7/0/ 0	22	0
		A	20	1251/ 85	911.7	9/0/ 1	22	0
PORTFL6	12/ 24/ 1/ 0	S	5	1217/125	4410.1	9/0/ 0	22	0
		A	5	991/110	3205.9	6/0/ 0	22	0
		S	10	1217/125	2344.1	9/0/ 0	22	0
		A	10	1182/120	1729.0	8/0/ 3	22	0
		S	20	1217/125	1489.4	9/0/ 0	22	0
		A	20	1495/108	1080.3	11/0/ 6	22	0

TABLE 5.3

CUTEr problems with an artificial time delay, testing synchronus and asynchronous implementations on 5, 10, and 20 processors.

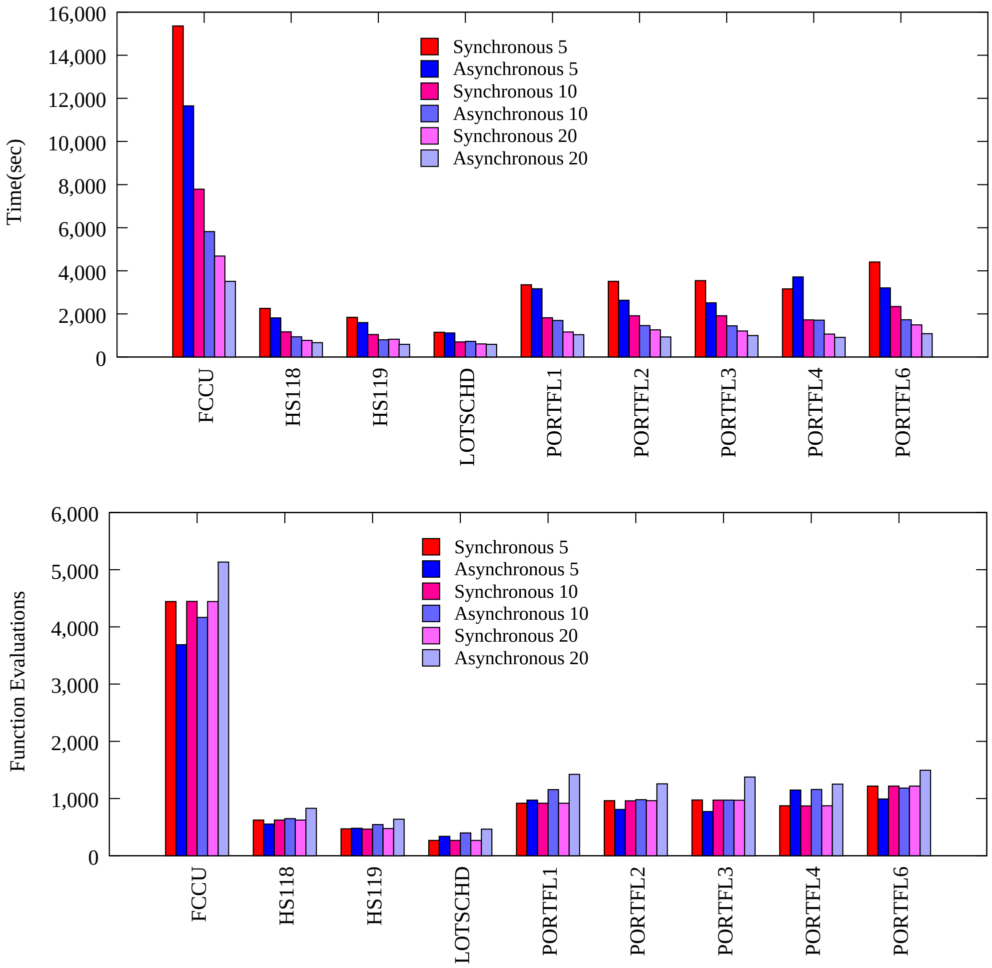


Fig. 5.2. *Comparisons of wall clock time (top) and function evaluations (bottom) for synchronous and asynchronous runs on 5, 10, and 20 processors.*

comparisons between asynchronous methods and synchronous methods based merely upon function evaluations do not tell the whole story.

Note that for the sake of time, to demonstrate this feature, we have used relatively low time delays, 5-15 seconds. In real life problems these time delays can be measured in minutes, hours, and even days.

5.4.3. Group 3: 101–1000 Variables. Though the primary focus of our numerical section is on the subset of the CUTEr test problems with 100 variables or less, we did explore the possibility of solving even larger problems. In this case, we were able to solve 11 (48%) of the 23 problems. On problem STATIC3, both APPSPACK and SNOPT detected unboundedness when the objective value dropped below -10^9 and -10^{14} respectively. The remaining 10 problems had bounded objective values. On 5 of the bounded problems we did worse than SNOPT and on 5 problems we did as well with the largest of these having 505 variables and 1008 inequality constraints.

For the problems we could not solve, we suspect the issue depends largely on the effects of inadequate scaling — i.e., the different parameters are based on entirely different scales and cannot be directly compared. As a result, our check for feasi-

Problem	n/ m_b/ m_e/ m_i	$f(x^*)$	Soln. Acc.	F-Evals	F-Cached	Time(sec)	D-LAPACK	D-CDDLIB	D-Cached	D-MaxSize	D-Appends
AGG	163/ 0/ 36/ 452	*Failed — could not find initial feasible point*									
DUAL3	111/ 222/ 1/ 0	1.4e-1	-8e-8	253405	1172	204.6	200	1	154	262	52
GMNCASE1	175/ 0/ 0/ 300	2.7e-1	4e-7	406502	558	1189.0	283	0	13	373	3
GMNCASE2	175/ 0/ 0/1050	*Failed — empty tangent cone encountered*									
GMNCASE3	175/ 0/ 0/1050	*Failed — function evaluations exhausted*									
GMNCASE4	175/ 0/ 0/ 350	*Failed — empty tangent cone encountered*									
HYDROELM	505/1010/ 0/1008	-3.6e6	-3e-7	56315	4247	5238.3	286	1	3	1422	3
HYDROELS	169/ 338/ 0/ 336	-3.6e6	3e-12	9922	645	49.6	96	0	0	334	0
PRIMAL1	325/ 1/ 0/ 85	-3.5e-2	-8e-10	393127	9981	4886.2	79	0	585	1031	296
PRIMAL2	649/ 1/ 0/ 96	*Failed — scaling: iterates became infeasible*									
PRIMAL3	745/ 1/ 0/ 111	*Failed — scaling: iterates became infeasible*									
PRIMALC1	230/ 215/ 0/ 9	-1.1	-1	73846	2535	294.0	4	0	10	460	0
PRIMALC2	231/ 229/ 0/ 7	-2.3e3	-3e-1	637282	1036	1359.5	3	0	0	462	0
PRIMALC5	287/ 278/ 0/ 8	-1.3	-1	16954	919	209.5	2	0	0	574	0
PRIMALC8	520/ 503/ 0/ 8	*Failed — max wall-time hit*									
QPCBOEI1	384/ 540/ 9/ 431	*Failed — Function evaluations exhausted*									
QPCBOEI2	143/ 197/ 4/ 181	*Failed — scaling: iterates became infeasible*									
QPCSTAIR	467/ 549/209/ 147	1.4e7	-6e-1	475729	11636	8683.7	8	97	0	353	0
QPNBOEI1	384/ 540/ 9/ 431	*Failed — scaling: iterates became infeasible*									
QPNBOEI2	143/ 197/ 4/ 181	*Failed — scaling: iterates became infeasible*									
QPNSTAIR	467/ 549/209/ 147	*Failed — empty tangent cone encountered*									
SSEBLIN	194/ 364/ 48/ 24	7.9e7	-8e-1	853875	55858	1865.8	157	0	6	288	0
STATIC3	434/ 144/ 96/ 0	-1.0e9	-1	23363	0	449.1	0	1	0	768	0

TABLE 5.4
CUTEr problems with 100 or more variables, tested on 60 processors.

bility of the trial points fails because it depends on the scaling but we do not have complete scaling information (which was the case in all four failed problems) because bound constraints are not specified. In practice, we would rely on problem-specific information provided by the scientists and engineers.

The maximum number of function evaluations being exceeded was due to the curse of dimensionality; that is, the lower bound on $\kappa(\mathcal{G})$ from (3.2) drops to zero as the number of variables increases, meaning that the search directions do not do as well at spanning the feasible region. For example, in the case of unconstrained optimization, $\kappa(\mathcal{G})$ degrades like $1/\sqrt{n}$ [18]. However, we were able to solve problem `HYDROELS`, with 169 parameters, using only 9,922 function evaluations.

5.5. Improving performance of GSS. In this section we suggest some strategies to improve the performance of GSS, especially in cases where GSS fails to converge normally. As examples, we consider all CUTEr test problems with 100 or fewer variables that exit abnormally using the default algorithm parameter settings; this accounts for 11% of the problems.

Modifications were made to the following input parameters to improve performance: s, the scaling vector; Δ_{tol}, the final step tolerance; x_0, the initial point; and α, the sufficient decrease parameter. There are two additional parameters that can be

Problem ($n \le 10$)	**Rel. Diff.**
HS44	.13
HS44NEW	.13
Problem ($10 < n \le 100$)	**Rel. Diff.**
None	–
Problem ($n > 100$)	**Rel. Diff.**
SSEBLIN	1e-4

Table 5.5

Problems whose best and worst objective value, obtained from 10 separate asynchronous runs, had a relative difference greater than 10^{-5}.

used in specific situations: f_{tol}, the function value tolerance; and ϵ_{mach}, the tolerance on constraint feasibility. Both of these parameters are adjustable in APPSPACK. The function tolerance parameter f_{tol} is standard in optimization codes and allows the algorithm to exit if the current iterate satisfies $f(x_k) < f_{\text{tol}}$. This parameter defaults to $-\infty$ in APPSPACK. The parameter ϵ_{mach} is used to define constraint feasibility and defaults to 10^{-12}. Because of numerical round-off, any algorithm that uses null-space projections to remain feasible must, for practical purposes, permit small nonzero constraint violations.

By tuning these six parameters, APPSPACK exited normally for all problems in this category with results summarized in Table 5.6.

Problem	Fix	$f(x^*)$	Soln. Acc.	F-Evals / F-Cached	Time(sec)	D-LAPACK / D-CDDLIB / D-Cached	D-MaxSize	D-Appends
AVION2	new x_0, $\alpha = 10^4$	9.5e7	-1e-11	2196/ 124	2.7	0/4/ 6	72	0
DEGENLPA	$s \leftarrow 10^{-3}s$, $\Delta_{\text{tol}} \leftarrow 10^{-7}\Delta_{\text{tol}}$	3.1	-9e-4	1437/ 230	2.3	7/0/ 0	10	0
DEGENLPB	$s \leftarrow 10^{-3}s$, $\Delta_{\text{tol}} \leftarrow 10^{-7}\Delta_{\text{tol}}$	-31	-4e-4	1433/ 196	2.3	7/0/ 0	10	0
HS268	$s = \begin{bmatrix} 1 & 1 & 10 & 10 & 100 \end{bmatrix}^T$	4.5e-2	-4e-2	33674/7150	2.5	3/0/ 1	10	0
MAKELA	$f_{\text{tol}} = 10^{-10}$	0.0	-2e-16	4514/ 102	2.9	0/0/ 0	42	0
QPCBLEND	new x_0, $\epsilon_{\text{mach}} = 5 \times 10^{-9}$	0.0	-9e-3	17/ 0	2.6	0/1/ 0	107	0
QPNBLEND	new x_0, $\epsilon_{\text{mach}} = 5 \times 10^{-9}$	0.0	-8e-3	17/ 0	2.6	0/1/ 0	107	0
S268	$s = \begin{bmatrix} 1 & 1 & 10 & 10 & 100 \end{bmatrix}^T$	4.5e-2	-5e-2	35758/7685	3.7	3/0/ 0	10	0
SIPOW4	$\Delta_{\text{tol}} \leftarrow 10^{-1}\Delta_{\text{tol}}$	2.7e-01	-3e-10	125/ 43	4.3	5/1/ 0	11	0

Table 5.6

Fixes to CUTEr problems with 100 or fewer variables that had abnormal exits.

Scaling is, in general, a problem for derivative-free methods that do not estimate the gradient. This makes it the first place to look for improvements. Because both `HS268` and `S268` lacked bound constraints, no scaling was used (see §4.1). Ideally, an appropriate scaling vector can be determined from knowledge about the underlying application. Barring that, we can sample the objective functions around the initial point. Defining

$$d_i = \frac{1}{20} \sum_{k=-9}^{10} \left| f\left(x_0 + \frac{k}{10} e_i\right) - f\left(x_0 - \frac{k-1}{10} e_i\right) \right|,$$

yields $d \approx \left[2e4, 3e4, 4e3, 7e3, 30\right]$ for both problems. Thus $f(x)$ is most sensitive to changes in x_1 and x_2 and least sensitive to changes in x_5. Using the scale vector $s = \left[1, 1, 10, 10, 100\right]^T$ (the order or magnitude differences between the entries in d) reduces the number function evaluations from more than one million to less than 36,000 for both problems.

Decreasing the final step tolerance is normally used to increase solution accuracy; however, decreasing this parameter can also be beneficial when an empty tangent cone is encountered, as was the case for SIPOW4, DEGENLPA, and DEGENLPB. An empty tangent cone indicates that the corresponding ϵ-active constraints spans $\mathbb{R}^n$ as Figure 2.1(d) illustrates. Step 13 in Algorithm 1 implies that ϵ is bounded below by the smallest step size, δ_{k+1}, that is in turn bounded by $\Delta_{\rm tol}$. Hence, decreasing $\Delta_{\rm tol}$ relaxes this bound and can reduce the final number of ϵ-active constraints. For DEGENLPA and DEGENLPB, reducing $\Delta_{\rm tol}$ alone was insufficient; therefore, the scaling vector was also reduced so that even fewer constraints were identified as active.

Another option is to try different starting points. These can be generated, for example, by doing a sampling in parameter space and choosing the best feasible point. In the three problems where this proved successful, we obtained a different starting point by using a different technique to project the infeasible CUTEr point to the feasible region. An alternate feasible starting point is generated by "snapping" the original infeasible point to the boundary, as described in §4.3. This technique is not always guaranteed to produce a feasible point but did so for the three problems under consideration here.

Both QPNBLEND and QPCBLEND benefited from a different starting point. However, both runs still resulted in an error: no trial points. This typically occurs because the projection of the trial points onto the equality constraints is not exact enough. This can be remedied by increasing the parameter $\epsilon_{\rm mach}$, as described above, thereby loosening the required tolerance.

Problem AVION2 also benefits from a different starting point but still requires a large number of function evaluations. This is because the sufficient decrease tolerance is too loose and new iterates with only miniscule improvements in the function value are accepted. In this case, we observed that the function values were all relatively large and so increasing α would force more improvement for accepting new best points.

The final problem in this category is `MAKELA` where the tangent cone at the solution is degenerate, requiring over 2^{20} generators. This is fundamentally unrepairable for GSS; however, we can do a workaround. If the user happens to know the target objective value, then we can specify the function tolerance, $f_{\rm tol}$. Making this change for MAKELA enables the method to exit successfully.

The following problems had best and worst objective values (found over ten averaged runs) that differed by a value greater than 10^{-5}: `HS268` had a maximum difference of .003, `S268` had a maximum difference of .005, and `DEGENLPA` had a value of 3.06 for one run and 4.59 for the remaining nine runs.

6. Conclusions. We have presented an asynchronous generating set search algorithm for linearly constrained optimization that is provably convergent to first-order optimal points; furthermore, we have demonstrated an implementation that is effective on a wide range of CUTEr test problems. This paper serves to bridge the gap between existing synchronous GSS methods that support linear constraints [18, 24, 25] and asynchronous GSS methods that support bound constraints [11, 17]. The synchronous methods work with a single step size at each iteration and so need only consider one tangent cone at a time, though that tangent cone may change from iteration to iteration. The asynchronous methods for bound-constrained problems rely on the fact that, even though multiple step sizes are in play, a single fixed set of generators is sufficient in all situations, namely, the coordinate search directions. In this paper, we bridge the gaps between these two approaches. We develop a strategy to handle multiple tangent cones simultaneously by appending additional search directions when needed.

In addition to theoretical results, we have also provided practical implementation details that can impact overall efficiency and performance, including scaling, function caching, snapping to the boundary, augmenting search directions, and direction caching. All enhancements have been implemented in version 5 of APPSPACK. Additionally, we expect function and direction caching to be useful when supporting nonlinear constraints such as [19], where a sequence of related optimization problems is solved for the same set of linear constraints.

We have also provided an extensive numerical study of the ability of GSS methods to handle linear constraints, extending results in [24, 16]. To the best of our knowledge, this is the most extensive study of direct search methods for linearly-constrained optimization problems. In practice, GSS methods are typically applied to problems with 100 or fewer variables. On CUTEr test problem of this size, with default parameter settings, APPSPACK is able to solve 89% of the problems. With minor changes to the input parameters we were able to obtains solutions for the remainder of these problems. Thus the numerical results demonstrate the ability of GSS methods to reliably obtain (as theory predicts) optimal objective values on problem sizes that are typical in practice for this approach. It is worth noting that, while GSS methods appear be restricted by problem dimension, they are able to handle a large number of constraints. Furthermore, we have once again [15, 17] shown the benefits of the asynchronous approach, which nearly always reduces the overall execution time, in many cases by 25% or more.

Acknowledgments. The authors wish to thank Rakesh Kumar and Virginia Torczon for their invaluable insights and stimulating discussions. We also thank the referees and editor for their careful reading of the manuscript and valuable suggestions for improvement.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, PA, third ed., 1999.

[2] C. Audet and J. E. Dennis, Jr., *Analysis of generalized pattern searches*, SIAM J. Optimiz., 13 (2003), pp. 889–903.

[3] S. I. Chernyshenko and A. V. Privalov, *Internal degrees of freedom of an actuator disk model*, J. Propul. Power., 20 (2004), pp. 155–163.

[4] M. L. Chiesa, R. E. Jones, K. J. Perano, and T. G. Kolda, *Parallel optimization of forging processes for optimal material properties*, in NUMIFORM 2004: The 8th International Conference on Numerical Methods in Industrial Forming Processes, vol. 712 of AIP Conference Proceedings, 2004, pp. 2080–2084.

[5] A. Conn, N. Gould, A. Sartenaer, and P. Toint, *Convergence properties of an augmented Lagrangian algorithm for optimization with a combination of general equality and linear constraints*, SIAM J. Optimiz., 6 (1996), pp. 674–703.

[6] A. R. Conn, N. I. M. Gould, and P. L. Toint, *Trust-Region Methods*, SIAM, Philadelphia, PA, 2000.

[7] G. Croue, *Optimisation par la méthode APPS d'un problème de propagation d'interfaces (in French)*, master's thesis, Ecole Centrale de Lyon, France, 2003.

[8] K. Fukuda, *cdd and cddplus homepage*. Web page: `http://www.cs.mcgill.ca/~fukuda/soft/cdd_home/cdd.html`, McGill University, Montreal, Canada, 2005.
[9] P. E. Gill, W. Murray, and M. A. Saunders, *SNOPT: An SQP algorithm for large-scale constrained optimization*, SIAM Rev., 47 (2005), pp. 99–131.
[10] N. I. M. Gould, D. Orban, and Ph. L. Toint, *CUTEr and SifDec: a constrained and unconstrained testing environment, revisited*, ACM T. Math. Software, 29 (2003), pp. 373–394.
[11] G. A. Gray and T. G. Kolda, *Algorithm 856: APPSPACK 4.0: Asynchronous parallel pattern search for derivative-free optimization*, ACM T. Math. Software, 32 (2006), pp. 485–507.
[12] G. A. Gray, T. G. Kolda, K. L. Sale, and M. M. Young, *Optimizing an empirical scoring function for transmembrane protein structure determination*, INFORMS J. Comput., 16 (2004), pp. 406–418. Special Issue on Computational Molecular Biology/Bioinformatics.
[13] C. Hernández, *Stereo and Silhouette Fusion for 3D Object Modeling from Uncalibrated Images Under Circular Motion*, PhD thesis, Ecole Nationale Supŕieure des Télécommunications, France, May 2004.
[14] P. D. Hough, T. G. Kolda, and H. A. Patrick, *Usage manual for APPSPACK 2.0*, Tech. Report SAND2000-8843, Sandia National Laboratories, Albuquerque, New Mexico and Livermore, California, 2000.
[15] P. D. Hough, T. G. Kolda, and V. J. Torczon, *Asynchronous parallel pattern search for nonlinear optimization*, SIAM J. Sci. Comput., 23 (2001), pp. 134–156.
[16] M. Jacobsen, *Real time drag minimization with linear equality constraints*, Tech. Report TRITA-AVE 2005:44, Aeronautical and Vehicle Engineering, Kungliga Tekniska Högskolan, Stockholm, Sweden, Dec. 2005.
[17] T. G. Kolda, *Revisiting asynchronous parallel pattern search for nonlinear optimization*, SIAM J. Optimiz., 16 (2005), pp. 563–586.
[18] T. G. Kolda, R. M. Lewis, and V. Torczon, *Optimization by direct search: new perspectives on some classical and modern methods*, SIAM Rev., 45 (2003), pp. 385–482.
[19] T. G. Kolda, R. M. Lewis, and V. Torczon, *A generating set direct search augmented Lagrangian algorithm for optimization with a combination of general and linear constraints*, Tech. Report SAND2006-5315, Sandia National Laboratories, Albuquerque, New Mexico and Livermore, California, Aug. 2006.
[20] T. G. Kolda, R. M. Lewis, and V. Torczon, *Stationarity results for generating set search for linearly constrained optimization*, SIAM J. Optimiz., 17 (2006), pp. 943–968.
[21] T. G. Kolda and V. Torczon, *On the convergence of asynchronous parallel pattern search*, SIAM J. Optimiz., 14 (2004), pp. 939–964.
[22] T. G. Kolda and V. J. Torczon, *Understanding asynchronous parallel pattern search*, in High Performance Algorithms and Software for Nonlinear Optimization, G. Di Pillo and A. Murli, eds., vol. 82 of Applied Optimization, Kluwer Academic Publishers, Boston, 2003, pp. 316–335.
[23] M. A. Kupinski, E. Clarkson, J. W. Hoppin, L. Chen, and H. H. Barrett, *Experimental determination of object statistics from noisy images*, J. Opt. Soc. Am. A, 20 (2003), pp. 421–429.
[24] R. M. Lewis, A. Shepherd, and V. Torczon, *Implementing generating set search methods for linearly constrained minimization*, Tech. Report WM-CS-2005-01, Department of Computer Science, College of William & Mary, Williamsburg, VA, July 2005. Revised July 2006.
[25] R. M. Lewis and V. Torczon, *Pattern search methods for linearly constrained minimization*, SIAM J. Optimiz., 10 (2000), pp. 917–941.
[26] J. Liang and Y.-Q. Chen, *Optimization of a fed-batch fermentation process control competition problem using the NEOS server*, P. I. Mech. Eng. I-J. Sys., 217 (2003), pp. 427–342.
[27] G. Mathew, L. Petzold, and R. Serban, *Computational techniques for quantification and optimization of mixing in microfluidic devices*. Available at `http://www.engineering.ucsb.edu/~cse/Files/MixPaper.pdf`, July 2002.
[28] D. McKee, *A dynamic model of retirement in Indonesia*, Tech. Report CCPR-005-06, California Center for Population Research On-Line Working Paper Series, USA, Feb. 2006.
[29] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall, *The double description method*, in Contributions to Theory of Games, Volume II, H. W. Kuhn and A. W. Tucker, eds., Princeton University Press, 1953.
[30] S. O. Nielsen, C. F. Lopez, G. Srinivas, and M. L. Klein, *Coarse grain models and the computer simulation of soft materials*, J. Phys.-Condens. Mat., 16 (2004), pp. R481–R512.
[31] V. Torczon, *On the convergence of pattern search algorithms*, SIAM J. Optimiz., 7 (1997),

pp. 1–25.