SANDIA REPORT

SAND2006-4621
Unlimited Release
Printed August 2006

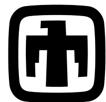
Asynchronous parallel generating set search for linearly-constrained optimization

Joshua D. Griffin, Tamara G. Kolda, and Robert Michael Lewis

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

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

Approved for public release; further dissemination unlimited.


Sandia National Laboratories

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

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

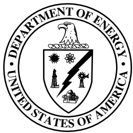
Printed in the United States of America. This report has been reproduced directly from the best available copy.

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

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

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

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online



SAND2006-4621
Unlimited Release
Printed August 2006

Asynchronous parallel generating set search for linearly-constrained optimization

Joshua D. Griffin and Tamara G. Kolda
Computational Science and Mathematics Research Department
Sandia National Laboratories
Livermore, CA 94551-9159
{jgriffi,tgkolda}@sandia.gov

Robert Michael Lewis
Department of Mathematics
College of William & Mary
Williamsburg, Virginia, 23187-8795
buckaroo@math.wm.edu

Abstract

Generating set search (GSS) is a family of direct search methods that encompasses generalized pattern search and related methods. We describe an algorithm for asynchronous linearly-constrained GSS, which has some complexities that make it different from both the asynchronous bound-constrained case as well as the synchronous linearly-constrained case. The algorithm has been implemented in the APPSPACK software framework and we present results from an extensive numerical study using CUTEr test problems. We discuss the results, both positive and negative, and conclude that GSS is a reliable method for solving small-to-medium sized linearly-constrained optimization problems without derivatives.

// Acknowledgments

The authors wish to thank Rakesh Kumar and Virginia Torczon for their invaluable insights and stimulating discussions.

Contents

1 Introduction 9
2 Asynchronous GSS for problems with linear constraints 11
2.1 Initializing the algorithm 14
2.2 Updating the search directions 14
2.3 Trial Points 16
2.4 Successful Iterations 16
2.5 Unsuccessful Iterations 16
2.6 An illustrated example 17
3 Theoretical properties 21
3.1 Definitions and terminology 21
3.2 Assumptions and conditions 23
3.3 Bounding a measure stationarity 25
3.4 Globalization 27
3.5 Global convergence 28
4 Implementation Details 31
4.1 Scaling 31
4.2 Function value caching 32
4.3 Snapping to the boundary 32
4.4 Generating conforming search directions 32
4.5 Direction caching 34
4.6 Augmenting the search directions 34
5 Numerical results 37
5.1 Test Problems 37
5.2 Choosing a starting point 37
5.3 Parameter Choices 38
5.4 Numerical results 38
6 Conclusions 49
References 51

Figures

1 Different set of conforming directions as x_k and ϵ_k vary. 15
2a Iteration $k = 0$ for example problem 18
2b Iteration $k = 1$ for example problem 18
2c Iteration $k = 2$ for example problem 18
2d Iteration $k = 3$ for example problem 19
2e Iteration $k = 4$ for example problem 19
2f Iteration $k = 5$ for example problem 19
3 Two options for additional search directions are the coordinate directions (left) or the normals to the linear inequality constraints (right). 35
4 Column descriptions for numerical results. 41
5 Comparisons of wall clock time (top) and function evaluations (bottom) for synchronous and asynchronous runs on 5, 10, and 20 processors. 45

Tables

1a CUTEr problems with 10 or fewer variables, tested on 20 processors. . . 39
1b CUTEr problems with 10 or fewer variables, tested on 20 processors. . . 40
2 CUTEr problems with 11–100 variables, tested on 40 processors. 43
3 CUTEr problems with an artificial time delay, testing synchronus and asynchronous implementations on 5, 10, and 20 processors. 44
4 CUTEr problems with 100 or more variables, tested on 60 processors. . 46
5 Problems whose best and worst objective value, obtained from 10 separate asynchronous runs, had a relative difference greater than 10^{-5}. . . 47

Algorithms

1 Asynchronous GSS for linearly-constrained optimization 12
2 Generating trial points 13
3 Sufficient decrease check 13

1 Introduction

Generating set search (GSS), introduced in [19], is a family of methods for derivative-free optimization that encompasses generalized pattern search [29, 2] and related methods. Two key features of GSS methods is that they can handle linear constraints and that they are easily parallelizable.

The problem of linear constraints for GSS has been studied by Kolda, Lewis, and Torczon [18], who present a GSS method for linearly-constrained optimization, and Lewis, Shepherd, and Torczon [22], who discuss the specifics of implementing 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 [23], which showed that the search directions must conform to the nearby boundary.

GSS methods have been parallelized in the software package APPSPACK [11, 15, 17], which is an asynchronous implementation of GSS for unconstrained and bound constrained problems and has proved to be useful in a variety of applications [3, 4, 7, 12, 13, 21, 24, 25, 26, 28]. The asynchronous implementation has the advantage of more efficiently using distributed resources by minimizing processor idle time. In numerical experiments, the asynchronous method has been as fast or faster than the synchronous method; for example, in recent work, the asynchronous method was 8–30% faster on a collection of benchmark test problems problems in well-field design [17].

In this paper, our contribution is to show how to handle linear constraints in an asynchronous GSS method. For GSS methods, the search directions must conform to the nearby boundary and the definition of "nearby" depends on the current step length control parameter. In the asynchronous implementation, different directions may have different step lengths, so we must handle that situation carefully.

The linearly-constrained optimization problem we consider is

$$\begin{array}{rc} \text{minimize} & f(x) \\ \text{subject to} & c_L \le A_I x \le c_U \\ & A_E x = b. \end{array} \tag{1}$$

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, along with a detailed discussion. 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 [18, 22]. Details that help

to make the implementation efficient are presented in §4, and we include extensive numerical results on problems from the CUTEr [10] test set in §5. We are able to solve problems with up to 505 variables and up to 2000 constraints. In fact, linear constraints often reduce the number of search directions at each iteration, thereby enabling us to solve larger problems than in the unconstrained case. We also compare synchronous and asynchronous versions of the code on several test problems, demonstrating that an asynchronous implementation can greatly reduce execution time. 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 [18] outline a GSS algorithm for problems with linear inequality constraints and consider both the simple and sufficient decrease cases. Lewis, Shepherd, and Torczon [22] extend this method to include linear equality constraints as well. 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. In addition to the parameters for the algorithm (discussed in §2.1), we assume that the user provides the linear constraints that define the feasible region, denoted by Ω, and a means for evaluating $f(x)$. The notation used is as follows. Subscripts denote the iteration index. The vector $x_k \in \mathbb{R}^n$ 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 $\mathcal{D}_k$. Superscripts denote the *direction index*, which ranges between 1 and $|\mathcal{D}_k|$ at iteration k. For simplicity in our discussions and because it is often practical, we assume

$$\| d_k^{(i)} \| = 1 \text{ for } i = 1, \ldots, |\mathcal{D}_k|. \tag{2}$$

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 this paper, we focus solely on the sufficient decrease case because it is the most practical. We present Algorithm 3 in terms of the forcing function

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

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.

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

1: $\mathcal{G}_0 \leftarrow$ generators for $T(x_0, \epsilon_0)$ where $\epsilon_0 = \min\{\delta_0, \epsilon_{\max}\}$
2: $\mathcal{D}_0 \leftarrow$ a set containing $\mathcal{G}_0$
3: $\Delta_0^{(i)} \leftarrow \delta_0$ for $i = 1, \ldots, |\mathcal{D}_0|$
4: $\mathcal{A}_0 \leftarrow \emptyset$
5: **for** $k = 0, 1, \ldots$ **do**
6: $\quad \mathcal{X}_k \leftarrow \{ \, x_k + \tilde{\Delta}_k^{(i)} d_k^{(i)} \; | \; 1 \leq i \leq |\mathcal{D}_k|, \;\; i \notin \mathcal{A}_k \, \}$ (see Alg. 2) ▷ generate trial points
7: $\quad$ send trial points $\mathcal{X}_k$ (if any) to the evaluation queue
8: $\quad$ collect a (non-empty) set $\mathcal{Y}_k$ of evaluated trial points
9: $\quad \bar{\mathcal{Y}}_k \leftarrow$ subset of $\mathcal{Y}_k$ that has sufficient decrease (see Alg. 3)
10: $\quad$ **if** there exists a trial point $y_k \in \bar{\mathcal{Y}}_k$ such that $f(y_k) < f(x_k)$ **then** ▷ successful
11: $\qquad x_{k+1} \leftarrow y_k$
12: $\qquad \delta_{k+1} \leftarrow \max\{\textsc{Step}(y_k), \Delta_{\min}\}$
13: $\qquad \mathcal{G}_{k+1} \leftarrow$ generators for $T(x_{k+1}, \epsilon_{k+1})$ where $\epsilon_{k+1} = \min\{\delta_{k+1}, \epsilon_{\max}\}$
14: $\qquad \mathcal{D}_{k+1} \leftarrow$ a set containing $\mathcal{G}_{k+1}$
15: $\qquad \Delta_{k+1}^{(i)} \leftarrow \delta_{k+1}$ for $i = 1, \ldots, |\mathcal{D}_{k+1}|$
16: $\qquad \mathcal{A}_{k+1} \leftarrow \emptyset$
17: $\qquad$ prune the evaluation queue to $q_{\max}$ or fewer entries
18: $\quad$ **else** ▷ unsuccessful
19: $\qquad x_{k+1} \leftarrow x_k$
20: $\qquad \mathcal{I}_k \leftarrow \{\textsc{Direction}(y) : y \in \mathcal{Y}_k \text{ and } \textsc{Parent}(y) = x_k\}$
21: $\qquad \delta_{k+1} \leftarrow \min \left\{ \tfrac{1}{2}\Delta_k^{(i)} \; \middle| \; i \in \mathcal{I}_k \right\} \cup \left\{ \Delta_k^{(i)} \; \middle| \; i \notin \mathcal{I}_k \right\}$
22: $\qquad \mathcal{G}_{k+1} \leftarrow$ generators for $T(x_{k+1}, \epsilon_{k+1})$ where $\epsilon_{k+1} = \min\{\delta_{k+1}, \epsilon_{\max}\}$
23: $\qquad \mathcal{D}_{k+1} \leftarrow$ a set containing $\mathcal{D}_k \cup (\mathcal{G}_{k+1} \setminus \mathcal{D}_k)$
24: $\qquad \Delta_{k+1}^{(i)} \leftarrow \begin{cases} \tfrac{1}{2}\Delta_k^{(i)} & \text{for } 1 \leq i \leq |\mathcal{D}_k| \text{ and } i \in \mathcal{I}_k \\ \Delta_k^{(i)} & \text{for } 1 \leq i \leq |\mathcal{D}_k| \text{ and } i \notin \mathcal{I}_k \\ \delta_{k+1} & \text{for } |\mathcal{D}_k| < i \leq |\mathcal{D}_{k+1}| \end{cases}$
25: $\qquad \mathcal{A}_{k+1} \leftarrow \{ \, i \; | \; 1 \leq i \leq |\mathcal{D}_k|, i \notin \mathcal{I}_k \, \} \cup \{ \, i \; | \; 1 \leq i \leq |\mathcal{D}_{k+1}|, \Delta_k^{(i)} < \Delta_{\text{tol}} \, \}$
26: $\quad$ **end if**
27: $\quad$ **if** $\Delta_{k+1}^{(i)} < \Delta_{\text{tol}}$ for $i = 1, \ldots, |\mathcal{D}_{k+1}|$ **then terminate**.
28: **end for**

Algorithm 2 Generating trial points

1: **for all** $i \in \{1, \ldots, |\mathcal{D}_k|\} \setminus \mathcal{A}_k$ **do**
2: $\quad \bar{\Delta} = \max\{\ \Delta > 0\ \mid\ x_k + \Delta d_k^{(i)} \in \Omega\ \}$ ▷ max feasible step
3: $\quad \tilde{\Delta} = \min\{\Delta_k^{(i)}, \bar{\Delta}\}$
4: $\quad$ **if** $\tilde{\Delta} > 0$ **then**
5: $\quad\quad y \leftarrow x_k + \tilde{\Delta} d_k^{(i)}$
6: $\quad\quad$ Step(y) $\leftarrow \Delta_k^{(i)}$
7: $\quad\quad$ Parent(y) $\leftarrow x_k$
8: $\quad\quad$ ParentFx(y) $\leftarrow f(x_k)$
9: $\quad\quad$ Direction(y) $\leftarrow i$
10: $\quad\quad$ add y to collection of trial points
11: $\quad$ **else**
12: $\quad\quad \Delta_k^{(i)} \leftarrow 0$
13: $\quad$ **end if**
14: **end for**

Algorithm 3 Sufficient decrease check

1: $\bar{\mathcal{Y}}_K \leftarrow \emptyset$
2: **for all** $y \in \mathcal{Y}_k$ **do**
3: $\quad \hat{f} \leftarrow$ ParentFx(y)
4: $\quad \hat{\Delta} \leftarrow$ Step(y)
5: $\quad$ **if** $f(y) < \hat{f} - \alpha\hat{\Delta}^2$ **then**
6: $\quad\quad \bar{\mathcal{Y}}_k \leftarrow \bar{\mathcal{Y}}_K \cup \{y\}$
7: $\quad$ **end if**
8: **end for**

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

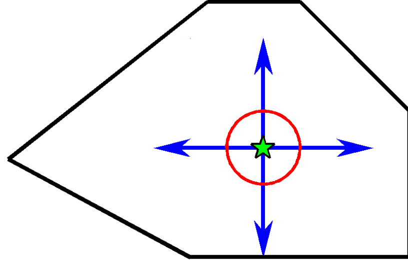
2.2 Updating the search directions

In Steps 1, 13, and 22, a set of conforming search directions, with respect to x and ϵ, is generated. In the synchronous algorithm, specifically, those are the directions that generate $T(x_k, \epsilon_k)$, the ϵ-tangent cone about x_k (see §3.1.1). The details of finding the generators are described in §4.4. Several examples of generating sets are shown in Figure 1. The choice of ϵ_k depends on Δ_k; specifically, we set $\epsilon_k = \min\{\Delta_k, \epsilon_{\max}\}$. The constant $\epsilon_{\max}$ provides a maximum distance for considering constraints because it generally does not make sense to consider constraints that are far away and can even confuse the method as seen in Figure 1(d). Asymptotically, however, $\epsilon_k = \Delta_k$.

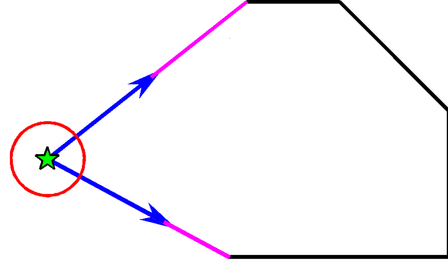
In the asynchronous case, meanwhile, every search direction has its own step length, $\Delta_k^{(i)}$. Consequently, $\mathcal{D}_k$, the set of search directions at iteration k, must contain generators for each of the following cones:

$$T(x_k, \epsilon) \text{ for all } \epsilon = \min\{\Delta_k^{(i)}, \epsilon_{\max}\} \text{ for } i = 1, \ldots, |\mathcal{D}_k|. \tag{3}$$

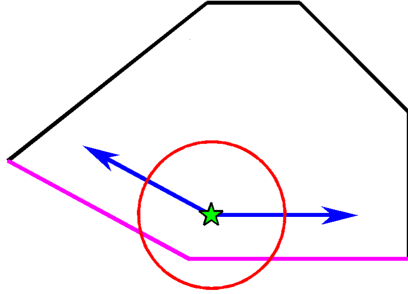
This requirement is not as onerous as it may at first seem. After successful iterations, the step sizes are all equal, so only one tangent cone is relevant (Step 13). It is only after an unsuccessful iteration that generators for multiple tangent cones may be needed simultaneously. As the individual step sizes $\Delta_k^{(i)}$ are reduced, which they



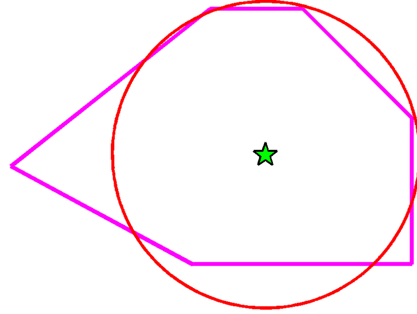
(a) The ϵ-ball does not intersect any constraints; any positive spanning set can be used.



(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.

Figure 1. Different set of conforming directions as x_k and ϵ_k vary.

will be by Theorem 3.12, 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 (3). 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. When the ϵ-active constraints do differ and different ϵ-tangent cones are produced, we generate the conforming directions for the new smaller value of ϵ_k in Step 22 and then merge them with the full direction set in Step 23. Even then, it is often the case that different values of $\epsilon_k^{(i)}$ yield identical sets of active constraints, so $\mathcal{D}_{k+1} = \mathcal{D}_k$.

2.3 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 $\textsc{Step}(y)$ stores $\Delta_k^{(i)}$ as opposed to the truncated step size $\tilde{\Delta}$; this prevents the step size from becoming prematurely small due to a point being near the boundary.

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.

2.4 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 the step that produced the new best point and $\Delta_{\min}$ (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].

2.5 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 §2.2.

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 suggests 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.

2.6 An illustrated example

In Figure 2, we illustrate six iterations of Algorithm 1, applied to the test problem

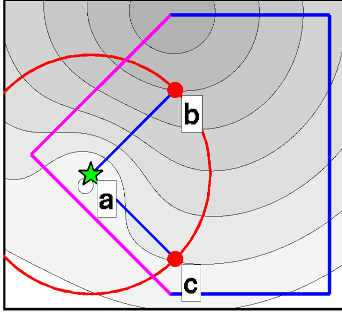
$$\begin{array}{ll} \text{minimize} & f(x) = \sqrt{9x_1^2 + (3x_2-5)^2} - 5\exp\left(\frac{-1}{(3x_1+2)^2+(3x_2-1)^2+1}\right) \\ \text{subject to} & \begin{array}{rcl} & 3x_1 & \leq 4 \\ -2 \leq & 3x_2 & \leq 5 \\ & -3x_1 - 3x_2 & \leq 2 \\ & -3x_1 + 3x_2 & \leq 5, \end{array} \end{array} \tag{4}$$

We initialize Algorithm 1 with $x_0 = \mathbf{a}$, $\Delta_{\text{tol}} = 0.01$ (though it's 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 2a. Shaded level curves illustrate the value of the objective function, with darker shades representing lower values. The feasible region is shown by the pentagon. The current best point, $x_0 = \mathbf{a}$, is denoted by a star. We calculate the 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 also initialize the step lengths, generating the trial points $\mathbf{b}$ and $\mathbf{c}$, both of which are submitted to the 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 2b shows the next iteration. The best point is updated to $x_1 = \mathbf{c}$. 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 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 2c, the best point is unchanged, i.e., $x_2 = x_1 = \mathbf{c}$. The value of δ_2 and hence ϵ_2 are reduced to $\frac{1}{2}$. In this case, however, the set of ϵ-active constraints is

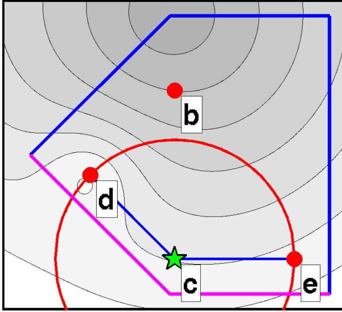


$x_0 = \mathbf{a}$, $\delta_0 = 1$, $\epsilon_0 = 1$
$\mathcal{D}_0 = \left\{ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \right\}$
$\Delta_0^{(1)} = \Delta_0^{(2)} = 1$
$\mathcal{X}_0 = \{\mathbf{b},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

Figure 2a. Iteration $k = 0$ for example problem

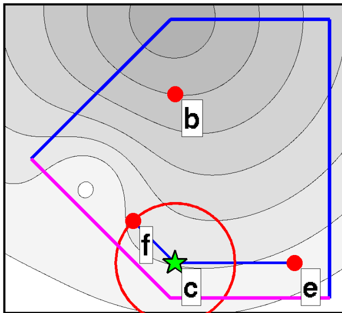


$x_1 = \mathbf{c}$, $\delta_1 = 1$, $\epsilon_1 = 1$
$\mathcal{D}_1 = \left\{ \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}$
$\Delta_1^{(1)} = \Delta_1^{(2)} = 1$
$\mathcal{X}_1 = \{\mathbf{d}, \mathbf{e}\}$, Queue $= \{\mathbf{b},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

Wait for evaluator to return...

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

Figure 2b. Iteration $k = 1$ for example problem

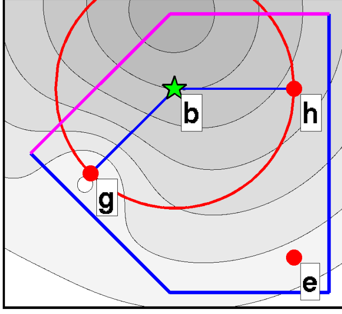


$x_2 = \mathbf{c}$, $\delta_2 = \frac{1}{2}$, $\epsilon_2 = \frac{1}{2}$
$\mathcal{D}_2 = \mathcal{D}_1$
$\Delta_2^{(1)} = \frac{1}{2}$, $\Delta_2^{(2)} = 1$
$\mathcal{X}_2 = \{\mathbf{f}\}$, Queue $= \{\mathbf{b}, \mathbf{e}, \mathbf{f}\}$

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

Figure 2c. Iteration $k = 2$ for example problem



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

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

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

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

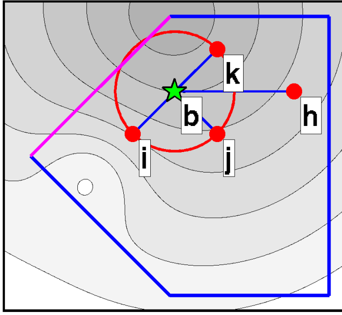
Wait for evaluator to return...

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

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

$\Rightarrow$ Unsuccessful

Figure 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 $= \{\mathbf{h}, \mathbf{i}, \mathbf{j}, \mathbf{k}\}$

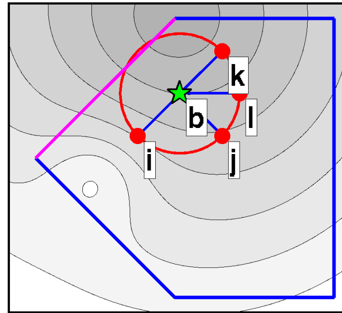
Wait for evaluator to return...

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

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

$\Rightarrow$ Unsuccessful

Figure 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...

Figure 2f. Iteration $k = 5$ for example problem

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, $\mathbf{f}$, is submitted to the queue. This time, two points return as evaluated, $\mathbf{f}$ and $\mathbf{b}$, the latter of which has the lower function value. In this case, we check that $\mathbf{b}$ satisfies sufficient decrease with respect to its parent, $\mathbf{a}$, and that it also satisfies simple decrease with respect to the current best point, $\mathbf{c}$. Both checks are satisfied, so the iteration is successful.

In Figure 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 $\mathbf{b}$. 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 $\mathbf{e}$ and $\mathbf{g}$ are returned, but neither satisfies sufficient decrease with respect to its parent. Thus, the iteration is unsuccessful.

In Figure 2e, the best point is unchanged, so $x_4 = x_3 = \mathbf{b}$. However, though our current point did not change, because $\delta_4 = \frac{1}{2}$ is reduced, $\epsilon_4 = \frac{1}{2}$ is also reduced. In contrast to iteration 2, the ϵ-active constraints have changed. The generators 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 $\mathbf{h}$ is returned, but it does not improve the function value, so the iteration is unsuccessful.

For Figure 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 points at iteration k may not be evaluated by iteration $k+1$. 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 in [18] are the concepts of tangent and normal cones. 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 [18], 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. The ϵ-tangent cone is its polar, i.e., $T(x,\epsilon) \equiv N(x,\epsilon)^\circ$.

We can form the 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 $\mathcal{S}$. For a given x and ϵ we can then define the index sets of ϵ-active constraints for A_I as

$$\begin{aligned}
\mathcal{E}_B &= \{ i : \; |\, (A_I)_i x - (c_U)_i \,| \leq \epsilon \| \, (A_I)_i \, \| \text{ and} \\
&\qquad\quad |\, (A_I)_i x - (c_L)_i \,| \leq \epsilon \| \, (A_I)_i \, \| \} \text{ (both)} \\
\mathcal{E}_U &= \{ i : \; |\, (A_I)_i x - (c_U)_i \,| \leq \epsilon \| \, (A_I)_i \, \| \} \setminus \mathcal{E}_B \text{ (only upper)} \\
\mathcal{E}_L &= \{ i : \; |\, (A_I)_i x - (c_L)_i \,| \leq \epsilon \| \, (A_I)_i \, \| \} \setminus \mathcal{E}_B \text{ (only lower)} \,,
\end{aligned}$$

and matrices V_P and V_L as

$$V_P = \begin{bmatrix} (A_I)_{\mathcal{E}_U} \\ -(A_I)_{\mathcal{E}_L} \end{bmatrix}^T \text{ and } V_L = \begin{bmatrix} A_E \\ (A_I)_{\mathcal{E}_B} \end{bmatrix}^T . \tag{5}$$

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 is 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 [18, 22, 19]. For any finite set of vectors $\mathcal{G}$, we define

$$\kappa(\mathcal{G}) \equiv \inf_{\substack{v\in\mathbb{R}^n\\ v_K\neq 0}} \max_{d\in\mathcal{G}} \frac{v^T d}{\|\, v_K \,\|\|\, d\,\|}, \text{ where } K \text{ is the cone generated by } \mathcal{G}. \tag{6}$$

It can be shown that $\kappa(\mathcal{G}) > 0$ if $\mathcal{G} \neq \{0\}$ [18, 23]. As in [18] we make use of the following definition:

$$\nu_{\min} = \min\{\kappa(\mathcal{V}) : \mathcal{V} = \mathcal{V}(x,\epsilon),\ x\in\Omega,\ \epsilon\geq 0,\ \mathcal{V}(x,\epsilon)\neq 0\}, \tag{7}$$

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 are 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 ([18]) *If $x \in \Omega$, then for all $\epsilon \geq 0$,*

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

where $\nu_{\min}$ is defined in (7).

3.1.2 A measure of stationarity

In our analysis, we use the first-order optimality measure

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

that has been used in previous analyses of GSS methods in the context of general linear constraints [19, 17, 18, 22]. 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 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 [18, 22], 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 contact $\kappa_{\min}$, 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 (6).

3.2.2 Conditions on the forcing function

Convergence theory for GSS methods typically requires either that all search directions lie on rational lattice or that a sufficient decrease condition be imposed [19, 18]. 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, i.e., ensuring that the step size ultimately becomes arbitrarily small [19, 18, 17]. Because it is both theoretically and computationally simpler than the alternative, we only consider the sufficient decrease case. 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

$$\rho(\Delta) = \frac{\alpha\Delta^2}{\beta + \Delta^2} \tag{8}$$

for $\alpha, \beta > 0$. The latter may offer some advantages because it is less restrictive on larger step sizes.

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 $\mathcal{F} = \{ x \in \Omega \mid f(x) \leq f(x_0) \}$ 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 [18] we combine Assumptions 3.4 and 3.6 to assert the existence of a constant $\gamma > 0$ such that

$$\| \nabla f(x) \| \leq \gamma, \tag{9}$$

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.

Condition 3.7 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 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, when *all* points corresponding to the ϵ-tangent cone have been evaluated and rejected. In this case, we can bound $\chi(x)$ in terms of the step size Δ_k [18]. 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). Furthermore, at iteration k suppose we have*

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

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

$$f(x_k) - f(x_k + \hat{\Delta}_k d) \ge \rho(\hat{\Delta}_k) \text{ for all } d \in \mathcal{G}. \tag{10}$$

Proof. Let $k^* \le k$ be the most recent successful iteration. Then $x_\ell = x_k$ for all $\ell \in \{k^*, \ldots, k\}$. Since $\hat{\Delta}_k \le \Delta_{\min}$, there exists $\hat{k}$ with $k^* \le \hat{k} \le k$ such that $\delta_{\hat{k}} = \hat{\Delta}_k$ in either Step 12 or Step 21 of Algorithm 1. Moreover, since $\hat{\Delta}_k \le \epsilon_{\max}$, we have $\epsilon_{\hat{k}} = \hat{\Delta}_k$ as well. Recalling $\mathcal{G}$ is the set of 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 $\hat{\Delta}_k$. And 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, (10) 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), satisfying Assumption 3.6 along with Conditions 3.2 and 3.3. If*

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

then

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

where the constant $\kappa_{\min}$ is from Condition 3.2 and the constant M is from Assumption 3.6.

Proof. Using Proposition 3.8, the proof is essentially the same as [18, Theorem 6.3]. Let $\mathcal{G}$ denote the set of generators for $T(x_k, \hat{\Delta}_k)$. By Condition 3.2 and (2), there exists a $\hat{d} \in \mathcal{G}$ such that

$$\kappa_{\min} \| [-\nabla f(x_k)]_{T(x_k,\hat{\Delta}_k)} \| \le -\nabla f(x_k)^T \hat{d}. \tag{11}$$

Proposition 3.8 ensures that

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

By the mean value theorem, there exists $\alpha \in (0, 1)$ such that

$$f(x_k + \hat{\Delta}_k \hat{d}) - f(x_k) = \hat{\Delta}_k \nabla f(x_k + \alpha \hat{\Delta}_k \hat{d})^T \hat{d}.$$

Thus,

$$\nabla f(x_k + \alpha \hat{\Delta}_k \hat{d})^T \hat{d} = \frac{f(x_k + \hat{\Delta}_k \hat{d}) - f(x_k)}{\hat{\Delta}_k} \ge -\frac{\rho(\hat{\Delta}_k)}{\hat{\Delta}_k}$$

Subtracting $\nabla f(x_k)^T \hat{d}$ from both sides and rearranging yields

$$-\nabla f(x_k)^T \hat{d} \le (\nabla f(x_k + \alpha \hat{\Delta}_k \hat{d}) - \nabla f(x_k))^T \hat{d} + \frac{\rho(\hat{\Delta}_k)}{\hat{\Delta}_k}.$$

Lipschitz continuity from Assumption 3.6 then implies

$$-\nabla f(x_k)^T \hat{d} \le M\hat{\Delta}_k + \frac{\rho(\hat{\Delta}_k)}{\hat{\Delta}_k}.$$

Combining this with (11) yields the desired result. □

The previous result involves a specific ϵ-tangent cone. The next result generalizes this to our desired use of the measure of stationarity $\chi(x_k)$, which is also bounded in terms of the step length Δ_k.

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

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

we have

$$\chi(x_k) \le \left(\frac{M}{\kappa_{\min}} + \frac{\gamma}{\nu_{\min}} \right) \hat{\Delta}_k + \frac{1}{\kappa_{\min}} \frac{\rho(\hat{\Delta}_k)}{\hat{\Delta}_k} \tag{12}$$

Proof. This proof follows [18, Theorem 6.4]. From Proposition 3.1 we have

$$\chi(x_k) \leq \left\| [-\nabla f(x_k)]_{T(x,\hat{\Delta}_k)} \right\| + \frac{\hat{\Delta}_k}{\nu_{\min}} \left\| [-\nabla f(x_k)]_{N(x,\hat{\Delta}_k)} \right\|$$

From Theorem 3.9, we have

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

The result follows the observation that

$$\left\| [-\nabla f(x_k)]_{N(x,\hat{\Delta}_k)} \right\| \leq \|\nabla f(x_k)\| \leq \gamma,$$

using the constant from (9). □

3.4 Globalization

Next, in Theorem 3.12, we show that the maximum step size can be made arbitrarily close to zero. This is the globalization of GSS methods [19]. 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 decrease 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 M, one can find a sequence of M 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) and that Algorithm 1 satisfies Condition 3.3 and Condition 3.7. Let $\mathcal{S} = \{k_1, k_2, \dots\}$ 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 $\hat{\mathcal{S}} = \{\ell_1, \ell_2, \dots\}$ 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

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

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

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

where $\hat{\Delta} = \text{Step}(x_{\ell_i})$. 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. □

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

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

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 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) satisfies Assumptions 3.4, 3.5, and 3.6 and Algorithm 1 satisfies Conditions 3.2, 3.3, and 3.7, then*

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

This page intentionally left blank.

4 Implementation Details

In this section we provide details of the implementation. For the most part we integrate the strategies outlined in [11, 14, 22].

4.1 Scaling

GSS methods are extremely sensitive to scaling, so it is important to use an appropriate scaling to get the best performance. As in [22], 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) to

$$\begin{aligned} \text{minimize} \quad & \hat{f}(\hat{x}) \\ \text{subject to} \quad & \hat{c}_L \leq \hat{A}_I \hat{x} \leq \hat{c}_U \\ & \hat{A}_E \hat{x} = \hat{b}, \end{aligned} \tag{13}$$

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:

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

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 will still use the notation in (1) but assume that the problem is appropriately scaled, i.e., as in (13).

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. The cached points are stored in a splay tree for efficient look-up. 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*, which defaults to $.5\Delta_{\text{tol}}$, and s_i is the scaling of the ith variable. For further details, see [14].

4.3 Snapping to the boundary

In Algorithm 2, we modify the step length so that we 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 somewhat subtle point is that if a function value cache like that in §4.2 is used, it may become impossible to evaluate certain points on the boundary if they lie within the cache tolerance setting of a previously evaluated point that is not on the boundary.

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

$$(A_I)_{\mathcal{S}} z = (c_I)_{\mathcal{S}} \tag{14}$$

Here (c_I) represents the appropriate lower or upper bound, whichever is active. We prune dependent rows from (14) so that the matrix has full row rank. LAPACK is then used to solve the generalized least squares problem $\| \, y - z \, \|$, subject to the constraint (14). If the solution z to the above least-squares problem is feasible for (1), then we reset $x = z$ before sending the trial point to the evaluation queue.

4.4 Generating conforming search directions

In Steps 1, 13, and 22, we have 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. We know that the total number of directions that will potentially be needed is finite (see [18]). Most problems (even degenerate ones) require a modest number of search directions; however, there are rare cases where the number of directions needed to generate the appropriate cone is quite large. Our numerical results in §5 verify these claims; the only problematic case was the problem MAKELA that required more than 2^{20} generators. In the nondegenerate case, the maximum number of generators needed at any single iteration is $2n$; moreover, adding linear constraints can only reduce the number of search directions (see Corollary 4.2).

Our method for generating appropriate conforming search directions follows [22]. Let V_P and V_L be formed as in (5). If the directions defining the normal cone are not degenerate, then the following theorem may be used.

Theorem 4.1 ([22]) *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 :*

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

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 a right inverse R exists for $V_P^T Z$. Then $T(x,\epsilon)$ is the positive span of the columns of $-ZR$ together with the linear span of the columns of ZN :

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

Thus whenever a right inverse for $V_P^T Z$ exists, we use the linear algebra software package LAPACK [1] to compute generators for $T(x,\epsilon)$. However, if $V_P^T Z$ fails to have a right inverse, signifying that the ϵ-active constraints are degenerate, we need to use a different method. In the degenerate case we use the C-library cddlib develop by Komei Fukuda [8], which implements the double description method of Motzkin et al. [27]. The following corollary follows immediately from Theorem 4.1 and a simple dimensionality argument.

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

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

In particular, if only $\epsilon-$active inequality constraints are active then

$$|\mathcal{G}_k| = 2n - r$$

where r equals the number of $\epsilon-$active inequality constraints.

Proof. We know that the magnitude of $\mathcal{G}_k$ is given by 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

$$\begin{aligned} n_z &= \dim(\text{null}(V_L^T)) \\ n_r &= \dim(V_p^T Z). \end{aligned}$$

Thus the total number of generators is given by

$$n_r + 2(n_z - n_r) = 2n_z - n_r.$$

The largest n_z can be however is n and the proof follows. □

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. 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.

Though the work required to compute generators is typically nominal compared to the costs of function evaluations, there are still occasions when the cost is non-trivial (possibly significantly so in the degenerate case). Moreover, cached directions can be reused across multiple optimizations when a sequence of objective functions are minimized for the same set of linear constraints. For example, the augmented Lagrangian approach in [20] requires such a sequence of solutions.

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. This can, however, make it difficult to approach optimal points that lie directly on nearby constraints. In order to allow boundary points to be approached directly, additional search directions may be added; two possible candidates for extra search directions are shown in Figure 3. In our experiments the (projected) constraints normals were added to the corresponding set of conforming search directions. That is, we append the columns of the matrix $(ZZ)^T V_P$, where Z and V_P are defined in Theorem 4.1.

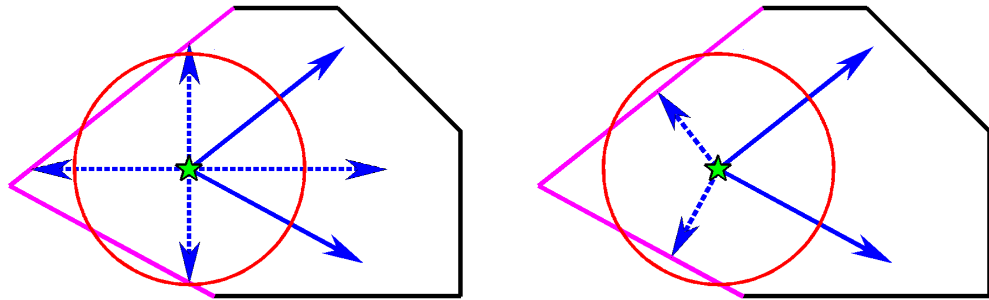


Figure 3. Two options for additional search directions are the coordinate directions (left) or the normals to the linear inequality constraints (right).

This page intentionally left blank.

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, 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. 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 have empty or single point 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 a majority of the problems, including problems with 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:

$$\begin{array}{ll} \text{minimize} & 0 \\ \text{subject to} & \begin{array}{rcl} c_L \leq & A_I x & \leq c_U \\ & A_E x & = b. \end{array} \end{array} \tag{15}$$

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

$$\begin{array}{ll} \text{minimize} & \| x - x_0 \|_2^2 \\ \text{subject to} & \begin{array}{rcl} c_L \leq & A_I x & \leq c_U \\ & A_E x & = b. \end{array} \end{array} \tag{16}$$

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 $\epsilon_{\text{snap}} = 0.5 \times 10^{-5}$, and we limited the number of function evaluations to 10^6. 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 1–4. Detailed descriptions of what each column indicates are shown in Figure 4. 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.

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 in 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 explicitly gives the differences for those cases.

5.4.1 Group 1: 1–10 Variables

Consider first Tables 1a and 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%).

Considering the solution accuracy, the final objective function obtained by APPSPACK was as good or better as that obtained by SNOPT, a derivative-based code. We compare against SNOPT only to illustrate 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

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	-4.6e+00	-6e-11	518/ 60	2.6	8/ 2/ 2	16	0
AVGASB	8/ 16/ 0/ 10	-4.5e+00	2e-11	462/ 51	2.3	8/ 0/ 0	16	0
BIGGSC4	4/ 8/ 0/ 13	-2.4e+01	0	117/ 11	1.7	4/ 0/ 0	8	0
BOOTH	2/ 0/ 2/ 0	*Failed — equality constraints determine solution*						
BT3	5/ 0/ 3/ 0	4.1e+00	-5e-11	145/ 38	2.3	1/ 0/ 0	4	0
DUALC1	9/ 18/ 1/214	6.2e+03	-5e-12	557/ 80	2.1	11/ 0/ 0	16	0
DUALC2	7/ 14/ 1/228	3.6e+03	-3e-12	304/ 40	2.8	10/ 0/ 0	12	0
DUALC5	8/ 16/ 1/277	4.3e+02	-5e-10	533/ 59	2.7	5/ 1/ 0	15	0
DUALC8	8/ 16/ 1/502	1.8e+04	-2e-11	488/ 56	3.7	9/ 0/ 4	14	0
EQC	9/ 0/ 0/ 3	*Failed — upper bound less than lower bound*						
EXPFITA	5/ 0/ 0/ 22	1.1e-03	-3e-10	1437/ 625	2.4	13/ 0/ 1	10	0
EXPFITB	5/ 0/ 0/102	5.0e-03	-5e-10	431/ 170	2.6	29/ 0/ 1	10	0
EXPFITC*	5/ 0/ 0/502	2.3e-02	-3e-08	3042/1584	5.4	20/36/277	31	135
EXTRASIM	2/ 1/ 1/ 0	1.0e+00	0	18/ 1	2.3	2/ 0/ 1	2	0
GENHS28	10/ 0/ 8/ 0	9.3e-01	-2e-10	168/ 47	4.2	1/ 0/ 0	4	0
HATFLDH	4/ 8/ 0/ 13	-2.4e+01	0	116/ 11	1.7	4/ 0/ 0	8	0
HIMMELBA	2/ 0/ 2/ 0	*Failed — upper bound less than lower bound*						
HONG	4/ 8/ 1/ 0	2.3e+01	-3e-10	191/ 36	2.0	4/ 0/ 1	6	0
HS105	8/ 16/ 0/ 1	1.0e+03	-1e-11	1606/ 199	3.3	6/ 0/ 0	16	0
HS112	10/ 10/ 3/ 0	-4.8e+01	-2e-09	2206/ 184	1.9	13/ 0/ 1	14	0
HS21	2/ 4/ 0/ 1	-1.0e+02	-8e-10	88/ 29	2.9	2/ 0/ 0	4	0
HS21MOD	7/ 8/ 0/ 1	-9.6e+01	-1e-16	1481/ 231	2.4	4/ 0/ 3	14	0
HS24	2/ 2/ 0/ 3	-1.0e+00	-4e-10	64/ 10	2.3	1/ 0/ 0	4	0
HS268	5/ 0/ 0/ 5	*Failed — evaluations exhausted*						
HS28	3/ 0/ 1/ 0	0.0e+00	0	152/ 50	2.7	1/ 0/ 0	4	0
HS35	3/ 3/ 0/ 1	1.1e-01	1e-10	180/ 34	2.2	1/ 0/ 0	6	0
HS35I	3/ 6/ 0/ 1	1.1e-01	-9e-10	136/ 34	2.6	1/ 0/ 0	6	0
HS35MOD	3/ 4/ 0/ 1	2.5e-01	0	87/ 4	2.3	3/ 0/ 1	4	0
HS36	3/ 6/ 0/ 1	-3.3e+03	0	88/ 2	3.5	3/ 0/ 0	6	0
HS37	3/ 6/ 0/ 2	-3.5e+03	-8e-11	135/ 24	2.6	1/ 0/ 0	6	0
HS41	4/ 8/ 1/ 0	1.9e+00	-5e-11	172/ 33	2.2	3/ 0/ 1	6	0
HS44*	4/ 4/ 0/ 6	-1.5e+01	0	137/ 11	3.0	7/ 0/ 0	8	0
HS44NEW*	4/ 4/ 0/ 6	-1.5e+01	0	132/ 12	2.6	4/ 0/ 0	8	0
HS48	5/ 0/ 2/ 0	0.0e+00	0	264/ 61	3.1	1/ 0/ 0	6	0
HS49	5/ 0/ 2/ 0	1.6e-07	-2e-07	25015/8392	4.8	1/ 0/ 0	6	0
HS50	5/ 0/ 3/ 0	0.0e+00	0	327/ 113	2.1	1/ 0/ 0	4	0

Table 1a. CUTEr problems with 10 or fewer variables, tested on 20 processors.

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	0.0e+00	0	132/ 31	2.7	1/ 0/ 0	4	0
HS52	5/ 0/ 3/ 0	5.3e+00	-9e-11	141/ 42	2.3	1/ 0/ 0	4	0
HS53	5/ 10/ 3/ 0	4.1e+00	-2e-09	127/ 43	2.7	2/ 0/ 1	4	0
HS54	6/ 12/ 1/ 0	-1.9e-01	2e-01	249/ 39	2.8	3/ 0/ 1	10	0
HS55	6/ 8/ 6/ 0	6.3e+00	5e-11	19/ 0	2.0	1/ 1/ 0	3	0
HS62	3/ 6/ 1/ 0	-2.6e+04	-9e-10	553/ 205	3.3	2/ 0/ 1	4	0
HS76	4/ 4/ 0/ 3	-4.7e+00	4e-11	253/ 38	2.3	5/ 0/ 0	8	0
HS76I	4/ 8/ 0/ 3	-4.7e+00	4e-11	233/ 47	2.5	6/ 0/ 0	8	0
HS86	5/ 5/ 0/ 10	-3.2e+01	-3e-11	156/ 23	3.1	6/ 1/ 0	11	0
HS9	2/ 0/ 1/ 0	-5.0e-01	0	60/ 5	2.2	1/ 0/ 0	2	0
HUBFIT	2/ 1/ 0/ 1	1.7e-02	-1e-10	84/ 18	2.8	2/ 0/ 0	4	0
LIN	4/ 8/ 2/ 0	-2.0e-02	-8e-07	22875/21193	5.4	1/ 2/ 1	6	0
LSQFIT	2/ 1/ 0/ 1	3.4e-02	-1e-10	84/ 20	2.2	2/ 0/ 0	4	0
ODFITS	10/ 10/ 6/ 0	-2.4e+03	1e-12	15818/ 4693	4.5	1/ 0/ 0	8	0
OET1	3/ 0/ 0/1002	5.4e-01	3e-10	754/ 168	4.0	0/ 19/ 0	8	0
OET3	4/ 0/ 0/1002	4.5e-03	-6e-07	1355/ 388	4.5	2/ 8/ 0	104	1
PENTAGON	6/ 0/ 0/ 15	1.4e-04	-3e-10	2205/ 527	2.2	6/ 0/ 0	12	0
PT	2/ 0/ 0/ 501	1.8e-01	4e-10	504/ 179	2.4	0/ 27/ 0	9	0
QC	9/ 18/ 0/ 4	-9.6e+02	4e-12	181/ 9	2.7	10/ 0/ 0	14	0
QCNEW	9/ 0/ 0/ 3	*Failed — upper bound less than lower bound*						
S268	5/ 0/ 0/ 5	*Failed — evaluations exhausted*						
SIMPLLPA	2/ 2/ 0/ 2	1.0e+00	0	439/ 111	3.0	2/ 0/ 0	4	0
SIMPLLPB	2/ 2/ 0/ 3	1.1e+00	0	388/ 97	2.9	3/ 0/ 0	4	0
SIPOW1	2/ 0/ 0/2000	-1.0e+00	0	171/ 324	4.0	0/151/ 0	6	0
SIPOW1M	2/ 0/ 0/2000	-1.0e+00	0	185/ 350	6.4	0/163/ 0	6	0
SIPOW2	2/ 0/ 0/2000	-1.0e+00	0	184/ 327	4.6	149/ 0/ 1	4	0
SIPOW2M	2/ 0/ 0/2000	-1.0e+00	0	178/ 327	4.7	149/ 0/ 0	4	0
SIPOW3*	4/ 0/ 0/2000	5.3e-01	-1e-10	760/ 188	5.4	70/ 15/ 0	459	1
SIPOW4	4/ 0/ 0/2000	*Failed — empty tangent cone encountered*						
STANCMIN	3/ 3/ 0/ 2	4.2e+00	0	69/ 24	2.7	3/ 0/ 0	6	0
SUPERSIM	2/ 1/ 2/ 0	*Failed — equality constraints determine solution*						
TAME	2/ 2/ 1/ 0	0.0e+00	0	51/ 22	2.8	2/ 0/ 0	2	0
TFI2	3/ 0/ 0/ 101	6.5e-01	0	713/ 175	2.0	36/ 0/ 0	6	0
TFI3	3/ 0/ 0/ 101	4.3e+00	7e-11	93/ 44	2.7	17/ 0/ 0	6	0
ZANGWIL3	3/ 0/ 0/ 3	*Failed — equality constraints determine solution*						
ZECEVIC2	2/ 4/ 0/ 2	-4.1e+00	-7e-10	66/ 29	2.1	1/ 0/ 0	4	0

Table 1b. CUTEr problems with 10 or fewer variables, tested on 20 processors.

- **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]:

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

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.

Figure 4. Column descriptions for numerical results.

different solutions on different runs on four problems (denoted by asterisks). 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., `LIN` 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 share similar traits with steepest descent methods; consequently, they 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 [22].

In general, the sets 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 time 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 of the entire course of the iterations (cf., `HS24/28/35`, 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., SIPOW1/1M/2/2M). It is important to have the capability to handle degenerate vertices; 13 (20%) 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 56 (85%) 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. Problems OET3 and SIPOW3 required 104 and 459 search directions, respectively, at a single iteration. The need to append search directions (D-Appends), which is unique to the asynchronous method, occurred in 16 (24%) cases.

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 4 cases, is like the situation shown in Figure 1(d). It can happen as a function of poor scaling of the variables when $\epsilon_{\max}$ is too large. The MAKELA is famously degenerate and requires $2^{20} + 1$ generators [22].

In two of the 16 problems, APPSPACK converged to different solutions across different runs. And on one of those two problems (KSIP), the solution was not as good as that obtained by SNOPT. Otherwise, all the solutions were comparable to that obtained by SNOPT.

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 four problems. The problem KSIP required 4126 search directions at one iteration. The problem DUAL1 required 249 appends to the search directions, indicating that it was near the solution for some time before it finally converged.

In Table 3, we compare synchronous and asynchronous runs of GSS; Corresponding bar graphs of the time and function evaluation comparisons are given in Figure 5. For these runs, artificial time delays have been added to simulate more expensive function evaluations. The time delay was selected randomly per evaluation to be between 5 and 15 seconds. We ran each problem on 5, 10, and 20 processors. Two things are worth noting here. One is that, in many cases, the asynchronous approach used a greater number of function evaluations. Second, despite evaluating more function values, the asynchronous approach took less time to solve the problem in every case

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-02	-2e-07	474829/2992	268.8	138/1/688	301	249
DUAL2	96/192/ 1/ 0	3.4e-02	-6e-08	176396/ 999	123.7	149/1/ 22	191	0
DUAL4	75/150/ 1/ 0	7.5e-01	-3e-08	56328/3584	32.3	91/1/ 15	283	1
FCCU	19/ 19/ 8/ 0	1.1e+01	-9e-11	4469/ 352	3.1	7/2/ 3	23	0
GOFFIN	51/ 0/ 0/ 50	0.0e+00	0e+00	13876/1339	6.0	2/0/ 0	102	0
HIMMELBI*	100/200/ 0/ 12	-1.7e+03	-8e-10	120273/2478	77.5	93/0/ 7	200	0
HIMMELBJ	45/ 0/ 14/ 0	*Failed — could not find initial feasible point*						
HS118	15/ 30/ 0/ 29	6.6e+02	-2e-16	634/ 64	2.7	24/0/ 0	36	0
HS119	16/ 32/ 8/ 0	2.4e+02	-3e-11	472/ 37	2.6	16/0/ 0	16	0
KSIP*	20/ 0/ 0/1001	1.0e+00	-3e-01	3161/ 124	142.0	2/4/ 0	4126	0
LOADBAL	31/ 42/ 11/ 20	4.5e-01	4e-09	53777/3189	11.2	13/0/ 0	40	0
LOTSCHD	12/ 12/ 7/ 0	2.4e+03	-1e-11	306/ 28	2.6	6/0/ 0	10	0
MAKELA4	21/ 0/ 0/ 40	*Failed — too many generators*						
NASH	72/ 0/ 24/ 0	*Failed — could not find initial feasible point*						
PORTFL1	12/ 24/ 1/ 0	2.0e-02	-3e-10	989/ 77	2.7	11/0/ 1	22	0
PORTFL2	12/ 24/ 1/ 0	3.0e-02	1e-09	879/ 85	2.7	6/0/ 1	22	0
PORTFL3	12/ 24/ 1/ 0	3.3e-02	4e-10	984/ 66	2.7	10/0/ 1	22	0
PORTFL4	12/ 24/ 1/ 0	2.6e-02	-1e-10	945/ 67	2.7	7/0/ 1	22	0
PORTFL6	12/ 24/ 1/ 0	2.6e-02	4e-09	984/ 70	3.6	8/0/ 1	22	0
QPCBLEND	83/ 83/ 43/ 31	*Failed — empty tangent cone encountered*						
QPNBLEND	83/ 83/ 43/ 31	*Failed — empty tangent cone encountered*						

Table 2. CUTEr problems with 11–100 variables, tested on 40 processors.

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	4445/348	15281.6	7/2/ 1	23	0
		A	5	3579/219	11199.8	16/1/ 8	23	0
		S	10	4442/351	7893.5	7/2/ 1	23	0
		A	10	4038/185	5627.1	25/1/58	23	0
		S	20	4441/352	4587.4	7/2/ 1	23	0
		A	20	4962/246	3277.1	16/1/61	22	0
HS118	15/ 30/ 0/ 29	S	5	616/ 73	2153.7	21/1/ 0	43	0
		A	5	536/ 76	1681.8	42/0/ 2	30	0
		S	10	616/ 73	1155.8	21/1/ 0	43	0
		A	10	645/ 94	913.1	47/0/ 2	30	0
		S	20	616/ 73	750.4	21/1/ 0	43	0
		A	20	783/158	588.9	48/0/ 1	30	0
HS119	16/ 32/ 8/ 0	S	5	466/ 33	1659.3	13/0/ 0	16	0
		A	5	476/ 39	1469.9	18/0/ 0	16	0
		S	10	466/ 35	982.7	13/0/ 0	16	0
		A	10	524/ 46	750.9	20/0/ 1	16	0
		S	20	472/ 36	780.6	13/0/ 0	16	0
		A	20	607/ 65	530.1	19/0/ 0	16	0
LOTSCHD	12/ 12/ 7/ 0	S	5	268/ 25	1103.8	6/0/ 0	10	0
		A	5	339/ 38	1081.9	6/0/ 8	10	0
		S	10	268/ 25	701.9	6/0/ 0	10	0
		A	10	395/ 39	673.5	7/0/10	10	0
		S	20	268/ 25	558.5	6/0/ 0	10	0
		A	20	465/ 46	568.0	7/0/11	10	0
PORTFL1	12/ 24/ 1/ 0	S	5	917/112	3237.8	9/0/ 2	22	0
		A	5	1014/ 95	3206.8	10/0/ 1	22	0
		S	10	918/111	1745.2	9/0/ 2	22	0
		A	10	1177/109	1642.3	10/0/ 2	22	0
		S	20	916/113	1136.4	9/0/ 2	22	0
		A	20	1545/114	1089.4	11/0/ 5	22	0
PORTFL2	12/ 24/ 1/ 0	S	5	960/114	3360.9	8/0/ 0	22	0
		A	5	807/ 90	2552.1	6/0/ 0	22	0
		S	10	960/114	1864.4	8/0/ 0	22	0
		A	10	978/ 85	1359.1	6/0/ 0	22	0
		S	20	962/112	1199.7	8/0/ 0	22	0
		A	20	1644/117	1142.8	10/0/ 4	22	0
PORTFL3	12/ 24/ 1/ 0	S	5	969/115	3419.3	11/0/ 0	22	0
		A	5	774/ 78	2444.2	7/0/ 0	22	0
		S	10	969/115	1849.6	11/0/ 0	22	0
		A	10	971/ 88	1362.2	8/0/ 1	22	0
		S	20	970/114	1189.0	11/0/ 0	22	0
		A	20	1402/ 91	984.1	13/0/ 6	22	0
PORTFL4	12/ 24/ 1/ 0	S	5	874/ 94	3131.1	7/0/ 0	22	0
		A	5	1135/109	3617.8	11/0/ 4	22	0
		S	10	874/ 94	1627.5	7/0/ 0	22	0
		A	10	1091/ 87	1539.2	11/0/ 4	22	0
		S	20	874/ 94	1059.8	7/0/ 0	22	0
		A	20	1214/ 82	871.1	8/0/ 0	22	0
PORTFL6	12/ 24/ 1/ 0	S	5	1215/127	4259.4	9/0/ 0	22	0
		A	5	987/108	3083.0	6/0/ 0	22	0
		S	10	1216/126	2309.1	9/0/ 0	22	0
		A	10	1204/121	1671.2	8/0/ 2	22	0
		S	20	1216/126	1434.9	9/0/ 0	22	0
		A	20	1418/102	996.6	10/0/ 6	22	0

Table 3. CUTEr problems with an artificial time delay, testing synchronus and asynchronous implementations on 5, 10, and 20 processors.

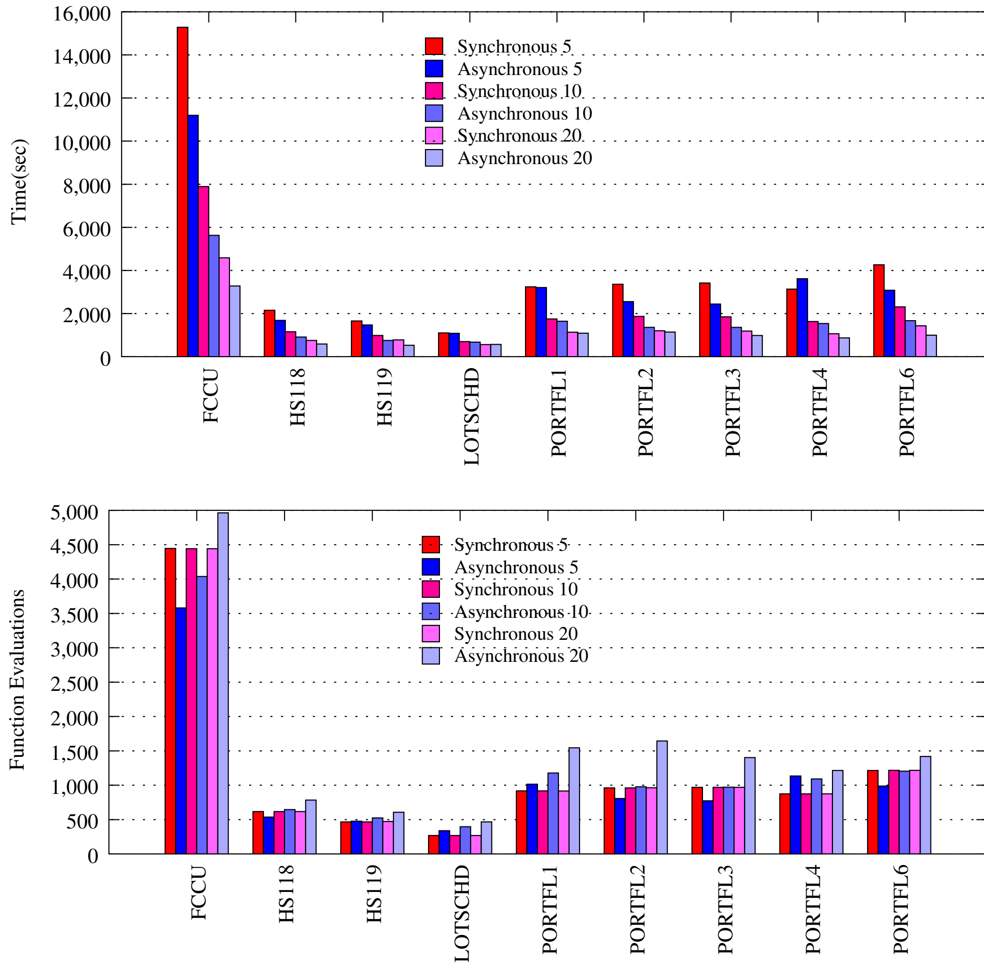


Figure 5. Comparisons of wall clock time (top) and function evaluations (bottom) for synchronous and asynchronous runs on 5, 10, and 20 processors.

save two (PORTFL4 on 5 processors and LOTSCHD on 20 processors). Thus the asynchronous approach was not only gaining more information in less time, but solving each problem in less time. This suggest that comparisons between asynchronous methods and synchronous methods based merely upon function evaluations may be almost irrelevant. 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

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-01	-8e-08	245923/ 1148	203.4	198/1/ 82	262	18
GMNCASE1	175/ 0/ 0/ 300	2.7e-01	4e-07	469060/ 1725	1398.9	282/0/106	538	49
GMNCASE2	175/ 0/ 0/1050	-9.9e-01	-1e-09	245306/ 522	2513.4	176/0/ 3	350	0
GMNCASE3*	175/ 0/ 0/1050	1.5e+00	-3e-09	374004/14820	12462.6	109/1/ 0	350	0
GMNCASE4	175/ 0/ 0/ 350	*Failed — empty tangent cone encountered*						
HYDROELM	505/1010/ 0/1008	-3.6e+06	-3e-07	55373/ 3512	4273.5	287/1/ 2	1422	2
HYDROELS	169/ 338/ 0/ 336	-3.6e+06	3e-12	9922/ 645	53.0	96/0/ 0	334	0
PRIMAL1	325/ 1/ 0/ 85	-3.5e-02	-8e-10	402563/10355	5108.1	82/0/592	1031	301
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	-1.1e+00	-1e-00	73774/ 2550	292.1	4/0/ 10	460	0
PRIMALC2	231/ 229/ 0/ 7	-2.3e+03	-3e-01	637764/ 1049	1417.4	3/0/ 0	462	0
PRIMALC5	287/ 278/ 0/ 8	-1.3e+00	-1e-00	16955/ 925	206.9	2/0/ 0	574	0
PRIMALC8	520/ 503/ 0/ 8	*Failed — max wall-time hit*						
QPCBOEI1	384/ 540/ 9/ 431	*Failed — scaling: iterates became infeasible*						
QPCBOEI2	143/ 197/ 4/ 181	*Failed — scaling: iterates became infeasible*						
QPCSTAIR	467/ 549/209/ 147	*Failed — scaling: iterates became infeasible*						
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 — scaling: iterates became infeasible*						
SSEBLIN*	194/ 364/ 48/ 24	7.9e+07	-8e-01	851858/47582	1824.9	157/0/ 7	307	0
STATIC3	434/ 144/ 96/ 0	*Failed — scaling: iterates became infeasible*						

Table 4. CUTEr problems with 100 or more variables, tested on 60 processors.

Problem ($n \leq 10$)	**Rel. Diff.**
EXPFITC	3e-4
HS44	.13
HS44NEW	.13
SIPOW3	.43
Problem ($10 < n \leq 100$)	**Rel. Diff.**
HIMMELBI	2e-5
KSIP	.29
Problem ($n > 100$)	**Rel. Diff.**
GMNCASE3	.54
SSEBLIN	.038

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

Though the primary focus of our numerical section is on the subset CUTEr test problem 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. However, four of those 11 did not have solutions that were as good as SNOPT was able to obtain. Of the remaining 7 problems, the largest had 505 variables and 1008 inequality constraints.

For the problems we could not solve, we supspect the issue depend 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 feasibility of the trial points fails because we are unable to appropriately perform this check. In general, we obtain scalings from the bound constraints. However, in the cases where we do not have complete scaling information (which was the case in all four failed problems), we must instead rely on problem-specific information, which would typically be provided for real-world applications, but is not available here.

In nearly all cases, the number of function evaluations was exceedingly large due to the curse of dimensionality. However, we were able to solve problem `HYDROELS`, with 169 parameters, using only 9,922 function evaluations.

This page intentionally left blank.

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 its effectiveness on a wide range of CUTEr test problems. This paper serves to bridge the gap between existing synchronous GSS methods that support linear constraints [19, 22, 23] and asynchronous GSS methods that support bound constraints [11, 17]. 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). 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 have bridged the gaps between these two approaches. We developed 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 have a huge impact on overall efficiency and performance, including scaling, function caching, snapping to the boundary, augmenting search directions, and direction caching. All of the enhancements have been implemented in APPSPACK. Beyond linear constraints, we expect that these features will also prove useful in solving the subproblems that typically arise in methods that support nonlinear constraints such as [20].

We have also provided an extensive numerical study of the ability of GSS methods to handle linear constraints, extending results in [22, 16]. To the best of our knowledge, this is the most extensive study of direct search methods for linearly-constrained optimization problems. The results demonstrate the ability to reliably obtain (as theory predicts) optimal objective values. 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.

This page intentionally left blank.

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, 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*. From McGill University, Montreal, Canada Web page: http://www.cs.mcgill.ca/~fukuda/soft/cdd_home/cdd.html, 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 8xx: APPSPACK 4.0: Asynchronous parallel pattern search for derivative-free optimization*, ACM T. Math. Software. To appear. Preprint at http://csmr.ca.sandia.gov/~tgkolda/ref#ACM-TOMS-APPSPACK4.

[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, 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, NM and Livermore, CA, 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, *Stationarity results for generating set search for linearly constrained optimization*, SIAM J. Optimiz. To appear. Preprint at http://csmr.ca.sandia.gov/~tgkolda/ref#SIAM-43363.

[19] ——, *Optimization by direct search: new perspectives on some classical and modern methods*, SIAM Rev., 45 (2003), pp. 385–482.

[20] ——, *Convergence properties of an augmented Lagrangian direct search algorithm for optimization with a combination of general equality and linear constraints*. In preparation, 2005.

[21] M. A. Kupinksi, 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.

[22] 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.

[23] R. M. Lewis and V. Torczon, *Pattern search methods for linearly constrained minimization*, SIAM J. Optimiz., 10 (2000), pp. 917–941.

[24] 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.

[25] 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.

[26] D. McKee, *A dynamic model of retirement in Indonesia*, Tech. Report CCPR-005-06, California Center for Population Research On-Line Working Paper Series, Feb. 2006.

[27] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall, *The double description method*, in Contributions to Theory of Games, H. W. Kuhn and A. W. Tucker, eds., vol. 2, Princeton University Press, 1953.

[28] 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.

[29] V. Torczon, *On the convergence of pattern search algorithms*, SIAM J. Optimiz., 7 (1997), pp. 1–25.

Sandia National Laboratories