ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

A Filter Active-Set Trust-Region (FASTr) Framework

Sven Leyffer

Mathematics and Computer Science Division

Technical Memorandum ANL/MCS-TM-298

October 1, 2007

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 1
- 1.2 Optimality Conditions . . . 2
- 1.3 Termination Conditions . . . 3

2 FASTr Algorithmic Framework **4**
- 2.1 FASTr Algorithm Outline . . . 4
- 2.2 Nonmonotone Filter for Global Convergence . . . 4
- 2.3 Subproblem Definition and Solvers . . . 6
- 2.4 Complete Algorithm Statement . . . 6

3 FASTr Interfaces and API **6**
- 3.1 Problem Structure . . . 8
- 3.2 Main Solver Interface . . . 8
- 3.3 User-Defined Functions . . . 8
  - 3.3.1 Sparse Jacobian Storage . . . 9
  - 3.3.2 Sparse Hessian Storage . . . 9

4 AMPL Options and Program Output **9**
- 4.1 AMPL FASTr Options . . . 10
- 4.2 Description of FASTr Output . . . 10
- 4.3 Description of FASTr Termination Conditions . . . 12

5 Future Extension of FASTr **12**

A Filter Active-Set Trust-Region (FASTr) Framework*

Sven Leyffer†

September 28, 2007

Abstract

We describe a sequential quadratic programming framework for solving nonlinear optimization problems. The framework is designed to be flexible and allows a range of subproblem solvers to be implemented. We describe the solver interfaces and an interface to the modeling language AMPL.

Keywords: Large-scale optimization, sequential quadratic programming, active-set methods, filter methods.

AMS-MSC2000: 90C20, 90C52.

1 Introduction

FASTr is an iterative Newton-type active-set framework for solving nonlinear optimization problems. This section defines the problem statement, optimality conditions, and termination conditions used in FASTr. The framework is written in C and is available upon request from the author. It requires access to the quadratic programming (QP) solver BQPD, which can be licensed from Roger Fletcher at the University of Dundee. Please be sure to obtain BQPD before requesting FASTr.

1.1 Problem Statement

FASTr is an iterative method for solving smooth nonlinear programs (NLPs) of the form

$$\begin{array}{lll} \underset{x}{\text{minimize}} & f(x) & \\ \text{subject to} & l_i^c \le c_i(x) \le u_i^c & \forall i = 1,\ldots,m_c \\ & l_i^a \le a_i^T x \le u_i^a & \forall i = 1,\ldots,m_a \\ & l_i^x \le x_i \le u_i^x & \forall i = 1,\ldots,m_x, \end{array} \tag{1.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $c_i : \mathbb{R}^n \to \mathbb{R}$ are the general nonlinear constraint functions, and $a_i^T x$ are the linear constrains. The bounds $l^c, u^c, l^a, u^a, l^x, u^x$ can be finite or infinite (larger than 1E20, or the value of `infty`; see Section 4). For maximization, simply multiply the objective by -1 (AMPL interface handles maximization automatically). FASTr is designed for smooth nonlinear optimization problems and assumes that the problem functions are twice continuously differentiable.

*Technical Memorandum ANL/MCS-TM-298.

†Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, `leyffer@mcs.anl.gov`.

In general, it may not be possible to find a feasible point for (1.1). In that case, our solver converges to a local minimum of the constraint violation, and this is taken as an indication that the constraints may be inconsistent. The feasibility problem has the form

$$
\begin{array}{lll}
\underset{x}{\text{minimize}} & \sum_{i \in J} w_i c_i(x) & \\
\text{subject to} & l_i^c \le c_i(x) \le u_i^c & \forall i = J^\perp \\
& l_i^a \le a_i^T x \le u_i^a & \forall i = 1, \ldots, m_a \\
& l_i^x \le x_i \le u_i^x & \forall i = 1, \ldots, m_x,
\end{array}
\tag{1.2}
$$

where $J, J^\perp$ is a partition of $\{1, \ldots, m_c\}$ into infeasible and feasible nonlinear constraints that is determined by the subproblem solver (we assume that the linear constraints are consistent, and note that this condition is checked at the start by the solver). The weights $w_i \in \{-1, 1\}$ allow the ready treatment of lower and upper bounds.

Notation: In the remainder, we indicate functions evaluated at a point $x^{(k)}$ by $f^{(k)} = f(x^{(k)})$. We denote a solution of (1.1) by x^* and use the notation $\nabla f^* = \nabla f(x^*)$.

1.2 Optimality Conditions

Our method aims to identify certain stationary points. In particular, we are interested in Karush-Kuhn-Tucker (KKT) points or Fritz-John (FJ) points if (1.1) is feasible. If, on the other hand, (1.1) is not feasible, then we are interested in KKT points of the feasibility problem (1.2) for some index sets $J, J^\perp$. A point x^* is a KKT point of (1.1) if and only if there exist multipliers λ, μ, ν such that the following conditions hold:

$$
\nabla f^* - \sum_{i=1}^{m_c} \lambda_i \nabla c_i^* - \sum_{i=1}^{m_a} \mu_i a_i^* - \nu = 0 \tag{1.3a}
$$

$$
l_i^c \le c_i^* \le u_i^c \perp \lambda_i \; \forall i = 1, \ldots, m_c \tag{1.3b}
$$

$$
l_i^a \le a_i^T x^* \le u_i^a \perp \mu_i \; \forall i = 1, \ldots, m_a \tag{1.3c}
$$

$$
l_i^x \le x_i^* \le u_i^x \perp \nu_i \; \forall i = 1, \ldots, m_x, \tag{1.3d}
$$

where $\perp$ represents complementarity, namely, (1.3b) means that

$$
\begin{array}{ll}
\lambda_i \ge 0, & \text{if } c_i^* = l_i^c \\
\lambda_i = 0, & \text{if } l_i^c < c_i^* < u_i^c \\
\lambda_i \le 0, & \text{if } c_i^* = u_i^c
\end{array}
$$

and (1.3c) and (1.3d) are defined similarly.

A point x^* is an FJ point of (1.1) if and only if there exist multipliers $\pi \ge 0, \lambda, \mu, \nu$ such that the following conditions hold:

$$
\pi \nabla f^* - \sum_{i=1}^{m_c} \lambda_i \nabla c_i^* - \sum_{i=1}^{m_a} \mu_i a_i^* - \nu = 0 \tag{1.4a}
$$

$$
l_i^c \le c_i^* \le u_i^c \perp \lambda_i \; \forall i = 1, \ldots, m_c \tag{1.4b}
$$

$$
l_i^a \le a_i^T x^* \le u_i^a \perp \mu_i \; \forall i = 1, \ldots, m_a \tag{1.4c}
$$

$$
l_i^x \le x_i^* \le u_i^x \perp \nu_i \; \forall i = 1, \ldots, m_x. \tag{1.4d}
$$

The FJ conditions (1.4) are a relaxation of the KKT conditions (1.3). The only difference is the existence of the additional objective multiplier π. Our solver will terminate at an FJ point if the constraints fails to satisfy a constraint qualification.

A point x^* is a stationary point of the feasibility problem (1.2) if and only if there exist multipliers λ, μ, ν such that the following conditions hold:

$$\sum_{i \in J} w_i \nabla c_i^* - \sum_{i \in J^\perp} \lambda_i \nabla c_i^* - \sum_{i=1}^{m_a} \mu_i a_i^* - \nu \;=\; 0 \tag{1.5a}$$

$$l_i^c \;\le\; c_i^* \;\le\; u_i^c \;\perp\; \lambda_i \;\forall i \in J^\perp \tag{1.5b}$$

$$l_i^a \;\le\; a_i^T x^* \;\le\; u_i^a \;\perp\; \mu_i \;\forall i = 1, \ldots, m_a \tag{1.5c}$$

$$l_i^x \;\le\; x_i^* \;\le\; u_i^x \;\perp\; \nu_i \;\forall i = 1, \ldots, m_x. \tag{1.5d}$$

We do not need to distinguish KKT and FJ points for the feasibility problem (1.2) because the subproblem solver will ensure the existence of multipliers (and otherwise lets us choose a different partition $J, J^\perp$).

1.3 Termination Conditions

The termination conditions for FASTr are linked to the optimality conditions of the previous section. In particular, FASTr terminates with an optimal solution when the following conditions are satisfied:

$$\|\nabla f^* - \sum_{i=1}^{m_c} \lambda_i \nabla c_i^* - \sum_{i=1}^{m_a} \mu_i a_i^* - \nu\| \;\le\; \epsilon \tag{1.6a}$$

$$\begin{aligned} &\sum_{i:\lambda_i<0} \lambda_i (c_i^* - u_i^c) + \sum_{i:\lambda_i>0} \lambda_i (c_i^* - l_i^c) \\ &+ \sum_{i:\mu_i<0} \mu_i (a_i^T x^* - u_i^a) + \sum_{i:\mu_i>0} \mu_i (a_i^T x^* - l_i^a) \\ &+ \sum_{i:\nu_i<0} \nu_i (x_i^* - u_i^x) + \sum_{i:\nu_i>0} \nu_i (x_i^* - l_i^x) \;\le\; \epsilon \end{aligned} \tag{1.6b}$$

$$\sum_{i=1}^{m_c} \| \max(l^c - c^* \;,\; c^* - u^c \;,\; 0)\|_1 \;\le\; \epsilon \tag{1.6c}$$

$$\sum_{i=1}^{m_a} \| \max(l^a - A^T x^* \;,\; A^T x^* - u^a \;,\; 0)\|_1 \;\le\; \epsilon \tag{1.6d}$$

$$\sum_{i=1}^{m_x} \| \max(l^x - x^* \;,\; x^* - u^x \;,\; 0)\|_1 \;\le\; \epsilon, \tag{1.6e}$$

where $A = [a_1, \ldots, a_{m_c}]$ are the linear constraint normals, ϵ is the user supplied tolerance `eps = 1E-6`, and the max in (1.6c) and so forth is taken componentwise. The residuals for the feasibility problem are defined similarly. We define the nonlinear constraint violation as

$$h(c(x)) \;:=\; \sum_{i=1}^{m_c} \| \max(l^c - c(x) \;,\; c(x) - u^c \;,\; 0)\|_1. \tag{1.7}$$

For a given partition $J, J^\perp$ of $\{1, \ldots, m_c\}$ we define the constraint violation as

$$h_J(c(x)) \quad := \quad \sum_{i \in J} \| \max(l^c - c(x) \ , \ c(x) - u^c \ , \ 0) \|_1, \tag{1.8a}$$

$$h_{J^\perp}(c(x)) \quad := \quad \sum_{i \in J^\perp} \| \max(l^c - c(x) \ , \ c(x) - u^c \ , \ 0) \|_1. \tag{1.8b}$$

It follows trivially that $h(c(x)) = h_J(c(x)) + h_{J^\perp}(c(x))$.

2 FASTr Algorithmic Framework

This section outlines the algorithmic framework implemented in FASTr, the underlying mechanisms that force global convergence, and the subproblem solvers. We first define the algorithmic framework in terms of the optimization problem (1.1). The feasibility restoration algorithm uses the same framework (and C code) to solve (1.2), and we indicate below how the definition of the filter has to change for this case.

2.1 FASTr Algorithm Outline

An outline of the FASTr framework is given below.

Outline of FASTr Framework

Given initial point $x^{(0)}$, trust-region radius ρ_0, set $k = 0$; compute $\nabla f^{(k)}$, $\nabla c^{(k)}$, etc.

while *not optimal* **do**

  Solve subproblem around $(x^{(k)}, \rho_k)$ for step $d^{(k)}$
  if $x^{(k)} + d^{(k)}$ *acceptable step* **then**
    Set $x^{(k+1)} := x^{(k)} + d^{(k)}$, and possibly increase $\rho_{k+1} = 2\rho_k$.
  else
    Set $x^{(k+1)} := x^{(k)}$, and decrease $\rho_{k+1} = \rho_k \ / \ 2$.

This outline leaves open two important questions that are answered in the remainder of this section: How do we decide to accept a step? How is the step computed? We note that our framework is very general. In particular, we allow multiple options for computing a step.

2.2 Nonmonotone Filter for Global Convergence

We promote global convergence of FASTr through the use of a trust region that controls the length of the step $d^{(k)}$, and a nonmonotone filter. The concept of a filter was introduced in [2, 3]. A filter promotes convergence to stationary points by viewing the optimization problem (1.1) as a bi-objective optimization problem in which both the objective, $f(x)$, and the nonlinear constraint violation, $h(c(x))$, are minimized.

A filter, $\mathcal{F}$, is a list of pairs of constraint violation and objective value, $(h^{(l)}, f^{(l)})$, such that no pair dominates another pair, namely, there exist no indices $l \in \mathcal{F}$ and $k \in \mathcal{F}$ such that

$$f^{(k)} < f^{(l)} \ \text{ and } h^{(k)} < h^{(l)}.$$

Dominance alone is not enough to ensure convergence because filter iterates may accumulate near an infeasible filter entry. To avoid this pitfall, we introduce a small margin around the filter, and we say that a point x is acceptable to the filter $\mathcal{F}$ if and only if

$$f(x) \le f^{(l)} - \gamma h(x), \quad \text{or } h(x) \le \beta h^{(l)}, \quad \forall l \in \mathcal{F}, \tag{2.1}$$

where $\beta = 0.999$ and $\gamma = 0.001$ are constants.

A nonmonotone filter acceptance criterion is defined by counting the number of filter entries that dominate a new entry in the sense of (2.1). To check whether a point x is acceptable, we define the index set

$$\begin{aligned} \mathcal{D} := \bigl\{ l \in \mathcal{F} : & \\ & f(x) > f^{(l)} - \gamma h(x) \text{ and } h(x) \ge \beta h^{(l)} \text{ or} \\ & f(x) \ge f^{(l)} - \gamma h(x) \text{ and } h(x) > \beta h^{(l)} \bigr\}. \end{aligned} \tag{2.2}$$

We say that a point x is acceptable to the nonmonotone filter $\mathcal{F}$ with memory $M > 0$ if and only if the cardinality $|\mathcal{D}| \le M$. Clearly, for $M = 0$, we recover the standard filter. The nonmontone filter has also been called a *shadow filter*. Figure 1 shows the sloping envelope of (2.1) that was introduced in [1] on the left and the new shadow filter on the right.

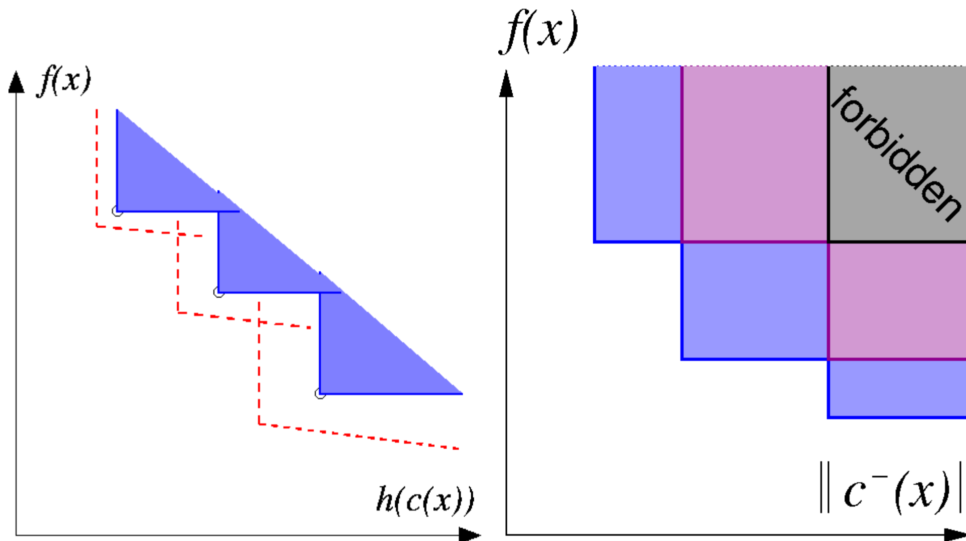

Figure 1: Sloping filter envelope (left) and nonmonotone filter with $M = 3$ (right).

A filter provides us only with feasible limit points, and a globally convergent algorithm also needs a sufficient reduction condition. We denote the solution of the QP subproblem by $q(d)$. Our sufficient reduction condition is applied only when we are sufficiently close to a feasible point that is measured through a switching condition. Formally, we require that

$$\Delta f := f^{(k)} - f^{(k+1)} \ge \sigma\bigl(q_k(0) - q_k(d)\bigr), \quad \text{whenever } \; q_k(0) - q_k(d) \ge \delta\bigl(h^{(k)}\bigr)^2, \tag{2.3}$$

where $q_k(d)$ is defined below and $\sigma = 0.1$ and $\delta = 0.999$ are constants. We also denote the predicted reduction as $\Delta q := q_k(0) - q_k(d)$.

The feasibility restoration phase uses a filter based on $h_J(c(x))$ and $h_{J^\perp}(c(x))$, defined in (1.8). The partition $J, J^\perp$ is determined by the subproblem solver and depends on the Phase I algorithm of the underlying subproblem solver. BQPD uses a one-at-a-time approach for Phase I, and this is reflected in the definition of $J, J^\perp$. Thus, our framework manages two different filters.

2.3 Subproblem Definition and Solvers

The step d is computed by solving a QP trust-region problem defined as

$$(\mathrm{TR}(x^{(k)}, \rho_k)) \left\{ \begin{array}{lll} \underset{d}{\text{minimize}} & {g^k}^T d + \frac{1}{2} d^T W^k d & \\ \text{subject to} & l_i^c \le c_i^k + \nabla {c_i^k}^T d \le u_i^c & \forall i = 1, \ldots, m_c \\ & l_i^a \le a_i^T x^k + a_i^T d \le u_i^a & \forall i = 1, \ldots, m_a \\ & l_i^x \le x_i^k + d_i \le u_i^x & \forall i = 1, \ldots, m_x, \\ \text{and} & \|d\|_\infty \le \rho_k, & \end{array} \right.$$

where ρ is the trust-region radius, and W^k is the Hessian of the Lagrangian. The trust-region problem may become infeasible, and we then switch to a feasibility restoration phase. The QP trust-region problem that is solved during the feasibility restoration is defined as

$$(\mathrm{FTR}(x^{(k)}, \rho)) \left\{ \begin{array}{lll} \underset{d}{\text{minimize}} & \sum_{i \in J} \nabla {c_i^k}^T d + \frac{1}{2} d^T W^k d & \\ \text{subject to} & l_i^c \le c_i^k + \nabla {c_i^k}^T d \le u_i^c & \forall i \in J^\perp \\ & l_i^a \le a_i^T x^k + a_i^T d \le u_i^a & \forall i = 1, \ldots, m_a \\ & l_i^x \le x_i^k + d_i \le u_i^x & \forall i = 1, \ldots, m_x, \\ \text{and} & \|d\|_\infty \le \rho_k, & \end{array} \right.$$

and the partition $J, J^\perp$ of $\{1, \ldots, m_c\}$ is determined by the QP solver BQPD. We note that $\mathrm{FTR}(x^{(k)}, \rho))$ is always consistent, by the definition of the sets $J, J^\perp$ and because we determine feasibility of the linear constraints at the first iteration.

Both problems are solved by the QP solver BQPD, and we refer to the documentation of that solver for more details. We are working to integrate other subproblem solvers and are developing a solver API that can be used to hook other subproblem solvers into our general framework.

2.4 Complete Algorithm Statement

The algorithm has an inner and an outer loop. The outer loop updates the current iterate $x^{(k)}$ and handles the transition between optimization and feasibility restoration phase. The inner loop computes an acceptable step by (repeatedly) solving the trust-region subproblem $(\mathrm{TR}(x^{(k)}, \rho_k))$ or $(\mathrm{FTR}(x^{(k)}, \rho))$, respectively, for a decreasing sequence of trust-region radii ρ. The complete algorithm is stated below.

The feasibility restoration uses the same algorithmic framework and code. To distinguish the two cases, we have a flag `RestPhase` in the code that changes the definition of the filter, when we are in the restoration phase. In particular, the framework uses two distinct filters: the optimality filter, referred to as $\mathcal{F}_O$ that stores the entries $(h^{(l)}, f^{(l)})$, and the restoration filter $\mathcal{F}_R$ that stores the filter entries during the restoration phase, namely, $(h_{J^\perp}^{(l)}, h_J^{(l)})$.

3 FASTr Interfaces and API

This section defines the interfaces to FASTr and the function and gradient calls that must be provided by the user. Details about the sparse storage of the gradients can be found in the BQPD manual.

FASTr: Filter Active-Set Trust-Region Method

Given x_0, ρ_0, and upper bound U; compute $\nabla f^{(k)}$, $\nabla c^{(k)}$, $W^{(k)}$
Set $k = 0$, and set `RestPhase=false`; initialize $\mathcal{F} := \mathcal{F}_O := \{(U, -\infty)\}$, and $\mathcal{F}_R = \emptyset$

while *not optimal* **do**
- reset trust-region radius $\rho \in [\underline{\rho}, \overline{\rho}]$
- **repeat**
  - **if** `RestPhase=false` **then**
    - solve optimality TR($x^{(k)}, \rho$) for the step d
  - **else**
    - solve feasibility FTR($x^{(k)}, \rho$) for the step d
    - **if** $J = \emptyset$ *and* $x^{(k)} + d$ *acceptable to* $\mathcal{F}_O$ **then**
      - set `RestPhase=false`; flush feasibility filter, $\mathcal{F}_R = \emptyset$, set $\mathcal{F} := \mathcal{F}_O$
      - **exit loop**
  - **if** $\exists$ *solution* d **then**
    - **if** $d = 0$ **then terminate** KKT point found
    - compute predicted reduction Δq
    - **if** `RestPhase=false` **then**
      - evaluate $f(x^{(k)} + d)$ and $h(c(x^{(k)} + d))$
    - **else**
      - evaluate $h_J(x^{(k)} + d)$ and $h_{J^\perp}(c(x^{(k)} + d))$
    - **if** $x^{(k)} + d$ *acceptable to* $\mathcal{F}$ *and* $x^{(k)}$ **then**
      - **if** $\Delta q^{(k)} < \delta (h^{(k)})^2$ **then**
        - set $\rho_k = \rho$, $d^{(k)} = d$, $\Delta q^{(k)} = \Delta q$, $\Delta f^{(k)} = \Delta f$
        - **if** `RestPhase=false` **then**
          - add $(h^{(k)}, f^{(k)})$ to $\mathcal{F}_O$ — *h-type iteration*
        - **else**
          - add $(h_J^{(k)}, h_{J^\perp}^{(k)})$ to $\mathcal{F}_R$ — *h-type iteration*
      - **else if** $\Delta f \geq \sigma \Delta q$ *and* $\Delta q \geq \delta (h^{(k)})^2$ **then**
        - set $\rho_k = \rho$, $d^{(k)} = d$, $\Delta q^{(k)} = \Delta q$, $\Delta f^{(k)} = \Delta f$ — *f-type iteration*
      - **else**
        - reduce trust-region radius $\rho = \rho/2$
    - reduce trust-region radius $\rho = \rho/2$
  - **else**
    - add $(h^{(k)}, f^{(k)})$ to $\mathcal{F}_O$; enter **restoration phase**:
    - set `RestPhase=true` and set $\mathcal{F}_R := \{(h_{J^\perp}^{(k)}, h_J^{(k)})\}$
- **until** *new* $x^{(k+1)}$ *found*
- set $k = k + 1$, update gradients $\nabla f^{(k)}$, $\nabla c^{(k)}$ & test for convergence

3.1 Problem Structure

The user must provide the bounds in the problem definition. The problem dimensions and bounds are defined in the following structure; see `ProblemStruct.h`.

```
typedef struct {
    long   n;                // number of variables
    long   m;                // number of constraints
    double *bl;              // lower bounds (1:n+m)
    double *bu;              // lower bounds (1:n+m)
    char *cstype;            // constraint type (linear/nonlinear)
} ProblemStruct;
```

The arrays `*bl` and `*bu` store the lower and upper bounds with the variables bounds l^x, u^x stored in the first `n` locations and the remaining bounds l^c, l^a, u^c, u^a stored in the remaining `m` locations (in any consistent order). The array `*cstype` is used to indicate which general constraints are linear/nonlinear. Set `*cstype[i-1]=0` to indicate that the ith general constraint is linear, and `= 1` otherwise.

3.2 Main Solver Interface

The main solver interface of FASTr has the following header, see `filter.c`.

```
long filter (long n, long m, long kmax, long iprint, FILE *nout,
          double* rho0, double fmin, double* f, double* h,
       double* x, double* y, double* c)
```

Here `kmax` is the maximum dimension of the null-space (see manual for BQPD), `iprint` is the print level (see below), `*nout` is the output file identifier, `*rho0` is the initial trust-region radius, and `fmin` is a lower bound on the objective function (the solver will terminate with an indication that the problem is unbounded, when a feasible point x with $f(x) <$ `fmin` is found. The parameters `*x` and `*y` contain the initial guess of the primal and dual variables on input (and the final optimal value on exit).

Upon exit, the parameters `*f`, `*h`, `*x`, `*y`, and `*c` contain the optimal values of $f(x^*)$, the final constraint violation $h(x^*)$, the optimal primal variables `*x`$=x^*$, the multipliers `*y`$=[\lambda, \mu, \nu]$, and the constraint bodies $[c(x), A^T x]$.

3.3 User-Defined Functions

The user must provide functions that evaluate the objective, constraint residuals, gradients, and Hessian. The headers of the problem functions are given below together with a description of the parameters; see `ProblemStruct.h`.

```
void objfun   (double *x, double *f, long *errflag);
void confun   (double *x, double *c, long *errflag);
void gradient (double *x, double *g, double *a, long *la, long *errflag);
void hessian  (double *x, double *y, long phase, double *ws, long *lws,
        long *errflag);
```

In all cases, `*x` stores the variables $x^{(k)}$ at which the functions are evaluated. The parameter `*f` is the objective function $f(x^{(k)})$, and `*c` are the constraint bodies $[c(x^{(k)}), A^T x^{(k)}]$. The objective

gradient is stored in `*g` in dense format (i.e., all n entries must be provided, even if zero). The Hessian routine also takes as input the current set of multipliers $[\lambda^{(k)} : \mu^{(k)}]$ and requires the user to store the Hessian in the parameters `*ws` and `*lws`. The sparse storage conventions are discussed below.

In all functions, `*errflag` is used to indicate function exceptions, such as IEEE exceptions. The algorithm has provisions for handling IEEE exceptions.

3.3.1 Sparse Jacobian Storage

The gradients of the objective and the constraints are stored in sparse column-formal in `*a` and `*la`. Given the matrix

$$\hat{A} = [g : A]$$

where g is the current gradient of $f(x)$ and A is the Jacobian of the constraints $c(x)$, the number of nonzeros in this matrix is `nnza`.

The matrix $\hat{A}$ contains gradients of the linear terms in the objective function (column 0) and the general constraints (columns 1:m). No explicit reference to simple bound constraints is required in $\hat{A}$. The information is set in the parameters `*a` and `*la`. In this sparse format, these vectors have dimension `a(1:nnza)` and `la(0:lamax)`, where `nnza` is the number of nonzero elements in $\hat{A}$ and `lamax` is at least `nnza+m+2`. The last `m+2` elements in `la` are pointers.

The vectors `a(.)` and `la(.)` must be set as follows: `a(j)` and `la(j)` for `j=1,nnza` are set to the values and row indices (respectively) of all the nonzero elements of $\hat{A}$. Entries for each column are grouped together in increasing column order. `la(0)` points to the start of the pointer information in `la`. `la(0)` must be set to `nnza+1` (or a larger value if it is desired to allow for future increases to `nnza`).

The last `m+2` elements of `la(.)` contain pointers to the first elements in the column groupings. Thus `la(la(0)+i)` for `i=0,m` is set to the location in `a(.)` containing the first nonzero element for column i of $\hat{A}$. Also `la(la(0)+m+1)` is set to `nnza+1` (the first unused location in `a(.)`). Note that `la(la(0)+1) = la(la(0)) + n` must hold to allow `n` locations to be stored in column 0 of $\hat{A}$.

3.3.2 Sparse Hessian Storage

The Hessian is stored in sparse-column format in `*ws` and `*lws`. Only the upper or lower triangle is stored (or any combination of it). Given the Hessian matrix $W^{(k)}$, the number of nonzeros is `nnzw`.

In this sparse format, these vectors have dimension `ws(1:nnzw)` and `lws(0:nnzw+n+2)`. The last `n+2` elements in `lws` are pointers. The vectors `ws(.)` and `lws(.)` must be set as follows: `ws(j)` and `lws(j)` for `j=1,nnzw` are set to the values and row indices (respectively) of all the nonzero elements of $W^{(k)}$. Entries for each column are grouped together in increasing column order.

`lws(0) = nnzw+1` points to the start of the pointer information in `lws`. The last `n+2` elements of `lws(.)` contain pointers to the first elements in the column groupings. Thus `lws(la(0)+i)` for `i=1,n` is set to the location in `ws(.)` containing the first nonzero element for column i of $W^{(k)}$. Also, `lws(lws(0)+n+1)` is set to `nnzw+1` (the first unused location in `ws(.)`).

4 AMPL Options and Program Output

This section describes the input options that are available from AMPL and the output and result produced by FASTr.

Table 1: AMPL options for FASTr.

Identifier	Description [Default Value].
`eps`	Tolerance for SQP solver [1E-6].
`fact`	Factor for upper bound on filter [1.25] (see `ubd` below).
`infty`	A large number [1E20] used for infinite bounds.
`iprint`	Synonym for `outlev` [0].
`kmax`	Dimension of null-space for BQPD [500].
`maxf`	Maximum filter length [50 fixed].
`maxiter`	Maximum number of iterations [1000].
`mxlws`	INTEGER workspace increment.
`mxws`	REAL workspace increment.
`objno`	Objective number: 1 = first, 0 = none [1].
`outlev`	Print level (0=silent, 3=verbose) [0].
`pname`	Problem name [NLPproblem].
`rho`	Initial trust region size [1E1].
`timing`	Indicator to time evaluations (0=no, 1=yes) [0].
`ubd`	Parameter for upper bound on filter [100]. The initial upper bound on the constraint violation of the filter is set to `max( ubd , fact*hc )`, where `hc` is the constraint violation at the starting point.

4.1 AMPL FASTr Options

We start by describing the AMPL options that are available in FASTr. The default options can be changed either by evoking the AMPL command or by setting the environment variable `fastr_options`. To change the print level (`outlev`) and the initial trust-region radius (`rho`) from the AMPL prompt, type

```
ampl: options fastr_options "outlev=2 rho=20.0";
```

where `ampl:` is the AMPL prompt and should not be typed. This changes the environment variable `fastr_options`, and the content is passed to FASTr. The environment variable can also be changed from the Linux command prompt, for example, by typing

```
export fastr_options="outlev=2 rho=20.0";
```

Table 1 lists the AMPL options, their description, and their default values.

4.2 Description of FASTr Output

A typical output of FASTr with `iprint=1` looks like the following:

```
Fastr Version 1.0
=================
   Number of variables   =         2; Number of constraints    =         0
   Major iteration limit =      1000; Minor iteration limit    =      1000
   Filter upper bound    =       100; Constraint violn factor  =      1.25
   Trust-region radius   =        10; Optimality tolerance     =     1e-06

Major Minor    TR-radius     StepNorm  Constraints    Objective Accept Phase
----------------------------------------------------------------------------
    0     0           10            0            0          909    0     2
    1     1           10         2.98            0         8.97    0     2
    2     2         1.24         1.24            0        8.023    0     2
    3     1         2.48       0.3796            0        6.504    0     2
    4     1         2.48       0.8604            0        5.705    0     2
    5     1         2.48      0.03386            0        4.942    0     2
    6     1         2.48     0.001433            0        4.941    0     2
    7     1         2.48    2.527e-06            0        4.941    1     2

---------------------------------------------------------------
Fastr Version 1.0: Solution Summary
===================================
  Ifail           =          0 ; Phase           =          2 ;
  Major iters     =          7 ; Minor iters     =          8 ;
  Objective value =      4.941 ; Constraint norm =          0 ;
  KKT-residual    =  9.444e-09 ; Complementarity =          0 ;
  Min. TR-radius  =       1.24 ; Max. TR-radius  =         10 ;
  Avg. TR-radius  =      4.806 ;
  Final step-norm =  2.527e-06 ; Final TR-radius =       2.48 ;
---------------------------------------------------------------

FASTr Version 1.0 (20070426):
Optimal solution found, objective = 4.941229317989186
Evals: obj = 9, constr = 9, grad = 8, Hes = 8
```

The header prints the current version number of FASTr, the problem size, and the values of some key parameters. For `iprint=1`, each major iteration prints a single line where the headers are defined in Table 2. The footer again prints the version number and a summary of the solution status, including the KKT residual, the number of iterations, the complementarity error, and the min/max/final trust-region radius. The final two lines are printed by AMPL and show the objective value, and the number of function evaluations.

If `Phase=2` in the output, then the columns correspond to the objective function and constraint violation of the original problem (1.1). If, however, `Phase=1`, then the columns correspond to the objective and the constraints of the feasibility problem (1.2), that is, `Objective` corresponds to $h_J(x)$, and `Constraints` corresponds to $h_{J^\perp}(x)$.

Table 2: Print Headers of FASTr.

Header	Description
`Major`	Major iteration index.
`Minor`	Number of minor iterations for this major iteration.
`TR`-radius	Trust-region radius.
`StepNorm`	Norm of the step.
`Constraints`	Value of the constraint violation, $h(c(x))$.
`Objective`	Value of the objective function, $f(x)$.
`Accept`	Indicates whether step was accepted (1) or rejected (0).
`Phase`	Optimization phase (phase=1 is feasibility restoration (problem (1.2)); phase=2 is optimization).

Table 3: Termination Conditions of FASTr.

`ifail`	Termination Condition
0	Optimal solution found.
1	Problem is unbounded below (feasible point found with $f(x) \leq$ `- infty`).
2	Linear constraints are inconsistent.
3	Nonlinear constraints are locally inconsistent.
4	Subproblem is locally inconsistent, but constraint violation is less than `eps`. Indicates failure of constraint qualification.
5	The trust-region radius became too small (less than `eps`).
6	Outer iteration limit reached.
7	Crash/IEEE error in user supplied routines (functions/gradients/Hessian).
8	Unexpected ifail from subproblem solver.
9	Not enough REAL workspace or parameter error.
10	Not enough INTEGER workspace or parameter error.
11	Cannot evaluate objective/constraints at the starting point.
12	Cannot evaluate gradients/Hessian at the starting point.
13	Unknown Error. Please contact the author.

4.3 Description of FASTr Termination Conditions

Upon termination, FASTr assigns an integer value to the parameter `ifail`. Table 3 describes how `ifail` relates to the termination conditions of FASTr.

We note that `ifail` between 0 and 4 are normal (successful) outcomes, in the sense that we cannot guarantee to globally solve the problem of finding a feasible point, or making sure that the limit point satisfies a constraint violation. Values of `ifail` that are greater than 4 correspond to failures of the method. However, `ifail=7, 11, 12` correspond to failures in the user supplied nonlinear functions. We note also that `ifail=5`, the trust-region converging to zero, frequently occurs if the gradients are wrong.

5 Future Extension of FASTr

The following is a list of planned extensions of the framework:

Addition of Other Subproblem Solvers. We will add hooks to other QP solvers to be used as subproblem solvers.

Extension to SLP-EQP Methods. We will develop sequential linear programming methods within our framework and accelerate these methods with equality QP steps.

Interfaces to Other Systems. We will develop interfaces to the COIN-OR API, to CUTEr, and to automatic differentiation packages.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. This work was also supported by the U.S. Department of Energy through the grant DE-FG02-05ER25694 and through NSF grant 0631622.

References

[1] C.M. Chin and R. Fletcher. On the global convergence of an SLP-filter algorithm that takes EQP steps. *Mathematical Programming*, 96(1):161–177, 2003.

[2] R. Fletcher and S. Leyffer. Nonlinear programming without a penalty function. *Mathematical Programming*, 91:239–270, 2002.

[3] R. Fletcher, S. Leyffer, and Ph. L. Toint. On the global convergence of a filter-SQP algorithm. *SIAM J. Optimization*, 13(1):44–59, 2002.

The submitted manuscript has been created by the UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne") under Contract No. DE-AC02-06CH11357 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.