A User's Manual for NIPSF: Nonlinear Interior-Point Solver, Fortran

Fredrik Saaf
September 1996 (Revised, October 1996)

TR96-24
1 Introduction

The present manuscript describes a Fortran implementation of the interior-point method for nonlinear programming problems (NLP). Theoretical aspects of the algorithm are treated briefly in Section 2, where we introduce notation and present the overall features of a globalized Newton interior-point algorithm. For a more detailed treatment of this material, we refer the reader to El-Bakry et al. [3]. Section 3 describes the package NIPSF (Nonlinear Interior-Point Solver, Fortran) and its features. Practical details essential to using the code are covered, such as compilation, input parameters and the appropriate format of user-supplied functions. We conclude with some numerical results in Section 4, where the NIPSF package is applied to test problems from the literature.

2 Theory

2.1 The General Nonlinear Programming Problem and the Lagrange Multiplier Framework

The nonlinear programming problem, which we will abbreviate NLP, is in its most general form the optimization of an objective function, denoted here by the symbol \( f(x) \), over some space \( x \in X \), subject to constraints on the variable \( x \). The literature on this subject is extensive (see for example [4, 11]).

For definiteness consider the following form of the NLP:

\[
\begin{align*}
\min_x & \quad f(x) \\
\text{s.t.} & \quad h(x) = 0, \\
& \quad g(x) \geq 0.
\end{align*}
\]

(1)

The functions \( f, h, g \) are defined so that

\[
\begin{align*}
f &: \mathbb{R}^n \rightarrow \mathbb{R}, \\
h &= (h_1, \ldots, h_{me})^T : \mathbb{R}^n \rightarrow \mathbb{R}^{me}, \quad me < n, \\
g &= (g_1, \ldots, g_{mi})^T : \mathbb{R}^n \rightarrow \mathbb{R}^{mi},
\end{align*}
\]

where \( n \) denotes the dimension of the problem, \( me \) the number of equality constraints and \( mi \) the number of inequality constraints.
A point \( x \) is called feasible if it satisfies the constraints. The solution of (1) (if one exists) is called optimal, and is denoted \( x^* \).

In preparation for developing necessary conditions for a candidate point to be optimal for the NLP, we introduce Lagrange multipliers \( y \in \mathbb{R}^{me}, z \in \mathbb{R}^{mi} \) and the Lagrangian function \( l : \mathbb{R}^{n+me+mi} \to \mathbb{R} \) defined thus:

\[
l(x, y, z) = f(x) + y^T h(x) - z^T g(x).
\] (2)

Note the sign convention for the multipliers \( z \), which will ensure that \( z_i \geq 0 \) in the developments that follow.

### 2.2 The Karush-Kuhn-Tucker (KKT) Conditions

Under reasonable assumptions, i.e., if some constraint qualification (see [11], [9]) holds, the Karush-Kuhn-Tucker, or KKT, conditions [8] must necessarily hold at an optimal point for the problem (1). Using the notation \( \Lambda(g) \) to denote a diagonal matrix with the vector \( g \) on its diagonal, these conditions can be stated

\[
\begin{align*}
\nabla_x l(x, y, z) & = 0, \quad \text{(3)} \\
h(x) & = 0, \quad \text{(4)} \\
g(x) & \geq 0, \quad \text{(5)} \\
z & \geq 0, \quad \text{(6)} \\
\Lambda(g)z & = 0. \quad \text{(7)}
\end{align*}
\]

The satisfaction of conditions (4)–(5) is obviously necessary for the feasibility of the candidate point. Equation (3) is referred to as optimality, and the relation (7) is known as complementarity. We note that the KKT conditions comprise a square set of \( n_{eq} = n + me + mi \) nonlinear equations, subject to the \( 2mi \) inequality constraints (5)–(6).

In the interior-point method, iterates must stay feasible with respect to the inequality constraints. This is a difficult task for general, nonlinear functions \( g(x) \). For this reason, it is common practice to introduce a set of slack variables \( s \in \mathbb{R}^{mi} \) and eliminate the need to be feasible with respect to a set of nonlinear constraints at the expense of instead solving a larger nonlinear system. The resulting equivalent slack-variable form of the KKT conditions
\[ \nabla_x l(x, y, z) = 0, \quad h(x) = 0, \quad g(x) - s = 0, \quad z \geq 0, \quad s \geq 0, \quad \Lambda(s)z = 0. \] (8) (9) (10) (11) (12) (13)

For convenience, we express the KKT conditions more compactly as
\[ F(x, y, z, s) = 0, \quad (z, s) \geq 0, \] (14) (15)

with
\[ F(x, y, z, s) \equiv \begin{pmatrix} \nabla_x l(x, y, z) \\ h(x) \\ g(x) - s \\ \Lambda(s)z \end{pmatrix}. \] (16)

The nonlinear system \( F(u) = 0 \) has dimension \( n_{eq} = n + me + 2mi \) in the unknowns \( u = (x, y, z, s) \in \mathbb{R}^{n_{eq}} \). In the practically important case of simple bounds, i.e., \( g(x) = x \), the introduction of slack-variables is unnecessary, and the linear system size can be reduced to \( n_{eq} = n + me + mi \).

### 2.3 The Interior-Point Method

A critically important feature of the interior-point method is the use of perturbed KKT conditions (PKKT), \( F_\sigma \), which result from perturbing the last \( mi \) components of \( F \) by \( \sigma > 0 \). The resulting expression is
\[ F_\sigma(x, y, z, s) = 0, \quad (z, s) \geq 0, \] (17) (18)

where
\[ F_\sigma(x, y, z, s) \equiv F(x, y, z, s) - \sigma \hat{e}, \] (19)

and the vector \( \hat{e} = (0, \ldots, 0, 1, \ldots, 1)^T \) has \( mi \) ones, i.e., the perturbation affects only the complementarity conditions.
An iterative method is in general necessary in order to obtain a solution to the PKKT conditions. We consider the standard Newton’s method which consists of the iterative solution of

\[
J(u_k) \Delta u_k = -F_\sigma(u_k), \\
u_{k+1} = u_k + \Delta u_k,
\]
given some initial guess \(u_0\). Here \(J\) denotes the Jacobian of \(F_\sigma\), i.e., the matrix of partial derivatives of \(F\) with respect to the independent variables,

\[
J_{i,j} \equiv \frac{dF_i}{du_j}.
\]  

(20)

Note that the Jacobian is independent of the perturbation \(\sigma\). A simple calculation shows that the Jacobian has the form

\[
J(u) = \begin{pmatrix}
\nabla^2_{x,x} f(x, y, z) & \nabla_x h(x) & -\nabla_x g(x) & 0 \\
\nabla_x h(x)^T & 0 & 0 & 0 \\
\nabla_x g(x)^T & 0 & 0 & -I \\
0 & 0 & \Lambda(s) & \Lambda(z)
\end{pmatrix},
\]  

(21)

where the gradients of vector-valued functions are defined

\[
(\nabla_x h(x))_{i,j} \equiv \frac{\partial h_j}{\partial x_i}.
\]  

(22)

The Hessian of the Lagrangian is given by

\[
\nabla^2_{x,x} l(x, y, z) = \nabla^2_{x,x} f(x) + \sum_{i=1}^{m_e} y_i \nabla^2_{x,x} h_i(x) - \sum_{i=1}^{m_i} z_i \nabla^2_{x,x} g_i(x),
\]  

(23)

2.4 The Globalized Interior-Point Algorithm

We are now in a position to define a globalized interior-point algorithm. In what follows, the symbol \(\| \cdot \|\) denotes the Euclidean (\(L_2\)) norm of a vector, the subscript \(k\) denotes an iteration counter and the parameters \(\lambda^1_k\), \(\lambda^2_k\) are real numbers satisfying \(0 < \lambda^1_k, \lambda^2_k < 1\).
Algorithm 1 The Globalized Newton Interior-Point Framework

0) Choose an initial guess \( u_0 = (x_0, y_0, z_0, s_0) \) satisfying \( (z_0, s_0) > 0 \).
   For \( k = 0, 1, \ldots \) do

1) Test for convergence: if \( \| F(u_k) \| < \epsilon \) exit.

2) Update perturbation parameter \( \sigma_k \).

3) Solve for perturbed Newton step, \( \Delta u_k = -(J(u_k))^{-1} F_{\sigma_k}(u_k) \).

4) Adjust step-length to ensure \( s_{k+1} > 0, z_{k+1} > 0 \).
   \[ \Delta u_k \leftarrow \lambda_k^1 \Delta u_k \]

5) Adjust step-length for globalization.
   \[ \Delta u_k \leftarrow \lambda_k^2 \Delta u_k \]

6) Update unknowns: \( u_{k+1} = u_k + \Delta u_k \).
   \( k \leftarrow k + 1 \)
   Goto 1

We now give a more detailed description of each step of the algorithm.

The initialization of the algorithm, Step 0, requires us to fix an appropriate initial guess \( u_0 \) for the primary unknowns and the Lagrange multipliers. The slack-variables \( s \) and the corresponding multipliers \( z \) must be chosen non-negative.

Any appropriate termination criterion (Dennis and Schnabel [2]) can be used in Step 1; we have implemented a simple test on the (absolute) \( L_2 \) error of the residuals (the KKT conditions) versus the user-specified tolerance \( \epsilon \). A condition on the relative error, such as

\[
\frac{\| F(u_k) \|}{1 + \| u_k \|} < \epsilon
\]

may be more appropriate in some circumstances.

Several possibilities exist for the implementation of Step 2; the selection of the perturbation parameter \( \sigma \). Numerical experimenting shows that an appropriate, systematic means of decreasing \( \sigma \) critically affects the convergence of the algorithm. Theoretical results are also known about “how fast”
this parameter must be made to approach zero in order to ensure rapid *local* convergence of the algorithm (El-Bakry et al. [3]).

We have implemented two approaches. The first is based on the notion of the "central path", which is the locus of points that are solutions to the PKKKT, $F_\sigma = 0$, traced out as $\sigma$ is varied (see Gonzalez-Lima [5] or Gonzalez-Lima et al. [6] for details in the Linear Programming case). In this mode, the algorithm will keep $\sigma$ fixed until the iterates are sufficiently close to satisfying the PKKKT for the current value of $\sigma$. This approach is close in spirit to the logarithmic barrier function formulation. However, rather than requiring that our iterates be within some tolerance of solving the PKKKT before reducing $\sigma$, we monitor only the satisfaction of the complementarity conditions, i.e., the condition used is

\[
\text{if } \|\Lambda(s_k)z_k - \sigma_k \hat{e}\| < \sigma_k \text{ then } \quad \sigma_k = \rho \sigma_k,
\]

where $0 < \rho < 1$. This choice was motivated by a desire to avoid "oversolving" by converging each sub-problem $F_\sigma(u) = 0$ to a high accuracy before reducing $\sigma$.

The second approach is a continuous variation of $\sigma$, based on the proximity to the boundary in $(s, z)$ space. It is similar to the approach described in El-Bakry et al. [3]. Specifically, we take

\[
\sigma_k = \min(\eta_1, \eta_2 s_k^T z_k) s_k^T z_k.
\]

Following El-Bakry et al. [3] we have chosen the values $\eta_1 = 0.2$ and $\eta_2 = 100$.

Step 3 entails the solution of a linear system which, in general, is not sparse. Currently, Linpack routines are used for this purpose, but any other module can be substituted if it is desired to exploit special structure of the problem. We note that in cases where no inequality constraints are present, the Jacobian matrix (21) is symmetric. For such cases the appropriate symmetric linear solvers are used.

In Step 4 we adjust the step length to ensure the non-negativity of the variables $s$ and $z$ at the new iteration level. This is in principle straightforward since the Newton update is linear. However, for reasons of numerical stability we do not allow a step that would extend the entire distance to the closest boundary, but choose instead to safeguard by requiring that we only move a certain fraction $0 < \tau_k < 1$ of the distance to the boundary. We point
out that to ensure the global convergence of the algorithm, additional tests are needed (El-Bakry et al. [3]). We find, however, that our approach works well for many practical problems. To retain the desired fast local convergence of the method, \( \tau_k \) must approach unity at a certain rate as we converge to a KKT point. To be specific, we take the reduction of the step to be

\[
\lambda_k^1 = \begin{cases} 
\min(1, -\tau_k \kappa_k^{-1}) & \text{if } \kappa_k < 0, \\
1 & \text{otherwise.}
\end{cases} \tag{26}
\]

Here, \( \kappa_k \) is given by

\[
\kappa_k = \min\left\{ \frac{\Delta s_k}{s_k}_1, \ldots, \frac{\Delta s_k}{s_k}_{mi}, \frac{\Delta z_k}{z_k}_1, \ldots, \frac{\Delta z_k}{z_k}_{mi} \right\}, \tag{27}
\]

and the fraction of movement to the boundary is

\[
\tau_k = \max(\tau_{\min}, 1 - s_k^T \Delta z_k). \tag{28}
\]

Directly following this reduction of the step length, another restriction on the step is imposed to satisfy the property of sufficient decrease in some appropriate measure. This is accomplished in Step 5 using a line-search strategy (see Dennis and Schnabel [2]) on the merit-function \( \phi_\sigma \) defined by

\[
\phi_\sigma(u) \equiv F_\sigma(u)^T F_\sigma(u), \tag{29}
\]

that is, the square of the \( L_2 \) norm of the residuals (the PKKT). This is a convenient choice of merit function, but many others are possible (cf. El-Bakry et al. [3] where the square of the \( L_2 \) norm of the KKT, i.e., \( \phi(u) \equiv F^T(u)F(u) \) is used, or Argaez et al. [1] and Parada et al. [10] for other choices).

The line-search procedure attempts to satisfy the Armijo-Goldstein condition (see [2]), i.e., to find the largest (positive) \( \lambda_k^2 < 1 \) such that

\[
\phi_\sigma(u_k + \lambda_k^2 \Delta u_k) < \phi_\sigma(u_k) + \alpha \lambda_k^2 \nabla \phi_\sigma(u_k)^T \Delta u_k, \tag{30}
\]

where the parameter \( \alpha \) satisfies \( 0 < \alpha < 1 \). Such a \( \lambda_k^2 \) can always be found (as long as the Jacobian is non-singular) since it is readily shown that the perturbed Newton step is a descent-direction for the merit function \( \phi_\sigma \), i.e.,

\[
\nabla \phi_\sigma^T \Delta u < 0. \tag{31}
\]
(A trivial calculation shows that $\nabla \phi_o^T \Delta u = -2\phi_o$). Two procedures for handling the backtracking line-search are implemented, namely interpolation and simple reduction. The interpolation algorithm uses standard cubic/quadratic interpolation (see Dennis and Schnabel [2]); the simple reduction scheme simply consists of shortening the step by a multiplicative factor.

3 Implementation

3.1 Overview

To improve readability, we use the following notational conventions throughout this section. Files on a Unix system are typeset in italics. Computer code variables are represented in a typewriter font and Fortran data types are in typewriter font with added square brackets, as in `<integer>`. Finally, the size of an array is indicated by the symbol `*`, e.g., `2*<integer>`.

The NIPSF package consists of Fortran subroutines for the solution of NLPs in the general form (1). These routines are compiled, using the supplied `Makefile`, into the library `libnipsf.a`. The user then links a call program (which is responsible for setting up storage and problem-specific parameter arrays, initial guesses and parameters needed by NIPSF) and user-supplied functions for evaluation of the functions involved in a particular application, with the library `libnipsf.a`. The sections below describe the parameters needed by NIPSF, the user-supplied code and the compilation procedure.

3.2 Parameters

A call to the NIPSF package has the form:

```fortran
    call nips(n, me, mi, iparam, niparam, rparam, nrparam,
      iwk, dimiwk, rwk, dimrwk, alpha, interpolate, nviol,
      hanalytic, reductionScheme, maxiter, tol, verbosity,
      checkSolution, solution, icount, returnFlag)
```

The arguments are now described in more detail.

`n` `<integer>
The number of unknowns ($n > 0$)."
me <integer>
The number of equality constraints ($0 \leq me < n$).

mi <integer>
The number of inequality constraints ($0 \leq mi$).

iparam niparam*<integer>
Problem-specific parameter array of integer type.

niparam <integer>
Size of the array iparam.

rparam nrparam*<real*8>
Problem-specific parameter array of double-precision type.

nrparam <integer>
Size of the array rparam.

iwk dimiwk*<integer>
Global integer work-space. Must be sufficiently large for memory allocation within the NIPSF package. This is checked.

dimiwk <integer>
Size of the integer work-space.

rwk dimrwk*<real*8>
Global double-precision work-space. Must be sufficiently large for memory allocation within the NIPSF package. This is checked.

dimrwk <integer>
Size of the double-precision work-space.

alpha <real*8>
The parameter used in the Armijo-Goldstein condition. Must be in the interval (0,1). Default value 1.0e-4.

interpolate <logical>
Use quadratic/cubic interpolation in line-search (.TRUE.) or a simple reduction strategy (.FALSE.).

nviol <integer>
Number of consecutive violations of the Armijo-Goldstein condition that we are willing to accept.
hanalytic  <logical>
Use user-supplied analytic Hessian (.TRUE.) or a finite-difference approximation of the Hessian (.FALSE.).

reductionScheme  <integer>
"Central-path" (reductionScheme = 1) or "Continuous" reduction of the perturbation parameter $\sigma$ (reductionScheme = 2).

maxiter  <integer>
Maximum number of iterations allowed.

tol  <real*8>
The absolute error-tolerance $\epsilon$ on the residuals of the KKT.

verbosity  <integer>
Verbosity of output. If verbosity = 0, only results are reported. If verbosity > 0, a convergence history is also generated.

checkSolution  <logical>
Check second-order sufficiency at the computed solution (.TRUE.).

solution  (n+me+2mi)*<real*8>
The computed solution (if successful). The $n$ primary variables, followed by $me$ equality multipliers, $mi$ inequality multipliers and $mi$ slack-variables.

itercount  <integer>
The number of iterations taken.

returnFlag  <integer>
Status of run. returnFlag = 0 indicates a successful run. If maxiter was exceeded, returnFlag = 1. If an exception occurs in line-search, returnFlag = 2.

3.3 User-supplied code

The user is required to include a main program which calls NIPSF, adhering to the calling-sequence specified above. The user must also include information about the functions that define the problem to be solved. These are the functions $f, h \in \mathbb{R}^{me}$ and $g \in \mathbb{R}^{mi}$ and their gradients $\nabla f \in \mathbb{R}^n$, $\nabla h \in \mathbb{R}^{n \times me}$ and $\nabla g \in \mathbb{R}^{n \times mi}$. In addition, the user may include code for
the analytic evaluation of the Hessian (by setting the flag \texttt{hanalytic} to the value \texttt{.TRUE.}). This, however, is optional, as the capability to approximate the Hessian using finite-differences is incorporated in the code.

We now proceed to specify the calling sequences used for the function evaluations. The functions \( f \), and, if constraints are present, \( h \) and \( g \) are evaluated in the Fortran subroutine \texttt{func}, which has the argument list

\[
\texttt{func}(f, h, g, x, n, \text{me}, \text{mi}, \text{iparam}, \text{niparam}, \\
\text{rparam}, \text{nrparam}, \text{rwk}, \text{dimrwk}, \text{iwk}, \text{dimiwk}).
\]

Note that regardless of the nature of the problem, \( h \) and \( g \) are always passed. This is done for reasons of uniformity. Although passed, they do not need to be defined if the corresponding constraint is not part of the problem (i.e., if \( \text{me} = 0 \) or \( \text{mi} = 0 \)).

Next, gradients must be provided as a Fortran subroutine \texttt{grad}. The calling sequence for evaluation of gradients is completely analogous, and has the form

\[
\texttt{grad}(\text{gradf}, \text{gradh}, \text{gradg}, x, n, \text{me}, \text{mi}, \text{iparam}, \text{niparam}, \\
\text{rparam}, \text{nrparam}, \text{rwk}, \text{dimrwk}, \text{iwk}, \text{dimiwk}).
\]

Finally, the optional analytic Hessian is provided, if desired, as the subroutine \texttt{hessian}. The calling sequence is

\[
\texttt{hessian}(\text{hessl}, x, y, z, n, \text{me}, \text{mi}, \text{iparam}, \text{niparam}, \\
\text{rparam}, \text{nrparam}, \text{rwk}, \text{dimrwk}, \text{iwk}, \text{dimiwk}).
\]

Note that the Lagrange multipliers \( y \) and \( z \) are explicitly included.

In all of the above code provisions have been made for solving more realistic problems by the explicit inclusion of integer and double-precision work-space arrays into all functions. This is particularly useful if function evaluations require interfacing with other software (such as running a forward simulation model in an inversion context). Also, in order to maintain clean interfaces, there is an explicit requirement that parameter arrays of both integer and real type be passed to any user-supplied function. This is a convenient way of communicating data in packed form without adding unnecessary clutter.
3.4 Compilation

The NIPSF package consist of the three sub-directories doc, where documenta-
tion can be found, source, which holds the source code and examples, which
contains a number of test-problems along with instructions for linking the
user-supplied code with the libnipsf.a library.

To build the library, proceed to the source directory, and issue the com-
mand:

make lib

This will create the library file libnipsf.a which can subsequently be linked
to the user’s code. To re-build from scratch, issue instead the command

make clean

and then

make lib

Once the library is built, it must be linked with the user-supplied routines
described above, and with the standard libraries Linpack, Blas and Lapack.

A typical linking statement would have the form:

f77 -o exe user.o libnipsf.a -llapack -llinpack -lblas

where we have assumed that all user code is collected in the file user.f. An
appropriate Makefile for this task can be found in the directory examples.

4 Numerical Examples

For the purpose of validation, we selected a suite of problems from Schit-
ttkowski and Hock [7]. All initial guesses were identical to those used in [7].
The results are collected in Table 1 below.
<table>
<thead>
<tr>
<th>Problem</th>
<th>n</th>
<th>me</th>
<th>mi</th>
<th>Iterations</th>
<th>Analytic Hessian</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>117</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>12</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>77</td>
<td>No</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>0</td>
<td>9</td>
<td>24</td>
<td>No</td>
</tr>
<tr>
<td>35</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>15</td>
<td>No</td>
</tr>
<tr>
<td>65</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>26</td>
<td>Yes</td>
</tr>
<tr>
<td>77</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>13</td>
<td>No</td>
</tr>
<tr>
<td>78</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>No</td>
</tr>
<tr>
<td>81</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>16</td>
<td>No</td>
</tr>
<tr>
<td>104</td>
<td>8</td>
<td>0</td>
<td>22</td>
<td>17</td>
<td>No</td>
</tr>
<tr>
<td>108</td>
<td>9</td>
<td>0</td>
<td>14</td>
<td>26</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1: Numerical results for a suite of Hock and Schittkowski test problems.

We make the following general observations.

- Convergence to the solutions reported in [7] was obtained in all cases but one (problem 108), where a distinct KKT point was found (however, the value of the objective function at the computed solution was the same in both cases). In problem 13, for which regularity (i.e., linear independence of the gradients of active constraints at the solution) does not hold, convergence was very sensitive to the selection of algorithmic parameters.

- Second-order sufficiency was tested, and the eigenvalues of the projected Hessian were found to agree with those reported in [7].

- More rapid convergence could often be achieved by relaxing the Armijo-Goldstein condition. For example, the problem 1 was solved in 20 iterations with $n_{viol} = 1$. 
• The finite-difference Hessian option was used where indicated with good results.

• In most cases, both the “continuous variation” and the “central-path” strategies for reducing $\sigma$ resulted in convergence with similar iteration counts.

5 Acknowledgments

The author wishes to thank Dr. Richard Tapia and Dr. Amr El-Bakry for many helpful discussions. A Matlab implementation of the interior-point algorithm for the general NLP, NIPS, developed by Dr. El-Bakry, served as a useful comparison in the initial development of NIPSF.

References


