User’s Guide for SQIC: Software for Large-Scale Quadratic Programming

Philip E. Gill† Elizabeth Wong‡
Department of Mathematics
University of California, San Diego
La Jolla, CA 92093-0112, USA

July 2, 2014

Abstract

SQIC is a software package for minimizing a general quadratic function subject to both equality and inequality constraints. SQIC is an implementation of an inertia-controlling active-set method that is capable of utilizing third-party linear solvers. It is efficient on problems with a small or large number of degrees of freedom.

SQIC is intended for large-scale problems with sparse Hessian and constraint matrices. The quadratic term \( \frac{1}{2}x^THx \) in the objective function is represented by either a user subroutine that returns the product \( Hx \) for a given vector \( x \), or a sparse matrix structure containing the matrix \( H \).

Keywords: optimization, large-scale quadratic programming, general quadratic programming, Fortran software.
## Contents

1. Introduction ......................................................... 3
   1.1 Implementation ............................................ 3
   1.2 Overview of the package .................................... 3
   1.3 Linear solvers ............................................... 4

2. Background .......................................................... 5
   2.1 Notation ...................................................... 5
   2.2 Active-set method ............................................ 5
   2.3 Schur complement and block LU methods .................... 6
   2.4 Variable-reduction method .................................. 6

3. Installing SQIC ....................................................... 7

4. Using SQIC ............................................................ 8
   4.1 Subroutines associated with SQIC ............................ 8
   4.2 Subroutine begin ............................................ 8
   4.3 Subroutine setSize .......................................... 8
   4.4 Subroutine setConstraints ................................... 9
   4.5 Subroutine setHessian ....................................... 9
   4.6 qpProb type ................................................ 10
   4.7 Subroutine SQIC ............................................. 12
   4.8 Subroutine end ................................................ 13
   4.9 Subroutine usrHx ............................................ 13

5. Optional parameters .................................................. 14
   5.1 The SPECS file ................................................. 14
   5.2 Subroutine specs ............................................. 14
   5.3 Multiple sets of options in the Specs file ................. 15
   5.4 Description of Optional Parameters ......................... 15

6. Interfaces ............................................................ 17
   6.1 Matlab interface ............................................ 17
   6.2 C interface .................................................. 17
   6.2.1 Subroutine begin .......................................... 17
   6.2.2 Subroutine specs .......................................... 18
   6.2.3 Subroutine sqic ........................................... 18
   6.2.4 Subroutine end ............................................ 19
   6.2.5 Subroutine usrHx .......................................... 19
   6.3 CUTEst interface .............................................. 20

References .............................................................. 21
1. Introduction

**SQIC** is a solver for *quadratic programs* (QP). It minimizes a quadratic objective function subject to bounds on the variables and linear constraints. The problem is assumed to be in the form

\[
\begin{align*}
\text{QP} & \quad \text{minimize} \quad \varphi(x) \\
& \quad \text{subject to} \quad \ell \leq \begin{pmatrix} x \\ Ax \end{pmatrix} \leq u,
\end{align*}
\]

where \( x \in \mathbb{R}^n \), \( \varphi(x) \) is a linear or quadratic objective function, \( \ell \) and \( u \) are constant lower and upper bounds, and \( A \) is an \( m \times n \) matrix.

In its most general form, the objective function has the form

\[
\varphi(x) = \phi + c^T x + \frac{1}{2} x^T H x,
\]

where \( \phi \) is a scalar constant, \( c \) is the constant linear objective vector, and \( H \) is the symmetric Hessian of the quadratic objective \( \varphi \). If \( H = 0 \), then the problem is a *linear program* (LP).

The bound constraints on \( Ax \) are converted to equality constraints by introducing a vector of *slack variables* \( s \). Internally, **SQIC** rewrites Problem QP in the following equivalent form

\[
\begin{align*}
\text{minimize} & \quad \varphi(x) \\
\text{subject to} & \quad Ax - s = 0, \quad \ell \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u.
\end{align*}
\]

1.1. Implementation

**SQIC** is implemented as a library of Fortran 2008 subroutines. A compiler capable of handling more recent additions to Fortran is required. Currently, suitable compilers include version 4.6 or higher of the GNU Fortran compiler *gfortran*, version 5.2 of the NAG compiler *nagfor*, or version 12.0 or higher of the Intel Fortran compiler *ifort*.

1.2. Overview of the package

A typical invocation of **SQIC** is

```fortran
use SQIC
...
type(qpProb) :: myProb
type(qpOpts) :: options
type(snInfo) :: output
...
call begin ( iPrint, iSumm, myProb, options )
call specs ( options, iSpecs, iExit )

m = ...
n = ...
ncObj = ...
neA = ...
nnH = ...
neH = ...
symmetric = .false.
```

! Allocates space for problem data
call setSize ( myProb, m, n, ncObj, Errors, options )
call setConstraints ( myProb, m, n, neA, Errors, options )
call setHessian ( myProb, nnH, neH, symmetric, Errors, options )

... ! The user should set up the problem here. ...

call solve ( 'Cold', myProb, output, iExit, options )
call end ( myProb )

The variable myProb is of type qpProb, the derived type for SQIC. The call to the begin subroutine initializes the Print and Summary files and the internal workspace, and must be called before calls to other SQIC routines. The call to specs is optional and allows the user to read in a set of run-time options from a file. The Hessian of the objective can be provided in two ways: in a user-defined subroutine usrHx or as a sparse-by-column structure. Space for the problem information is allocated by the calls to setSize, setConstraints, and setHessian. solve calls the main solver SQIC. end deallocates internal workspace.

More detailed descriptions of these subroutines and type structures are provided in Section 4.

1.3. Linear solvers

SQIC can use third-party linear solvers whose libraries must be provided by the user. The solvers are used to compute the LU factorization of a symmetric indefinite matrix and to solve systems involving the factors. Interfaces are provided for

- LUSOL (included in SQIC and is the default solver if no other solver is specified)
- HSL_MA57 [6, 10]
- HSL_MA97 [9, 10]
- SUPERLU [5]
- UMFPACK [1, 2, 3, 4]
2. Background

SQIC implements the nonbinding active-set method described in Gill and Wong [8] and Wong [11] for standard form problems (problems with linear equalities and simple bounds), restated here for convenience:

\[
\begin{align*}
\text{minimize} & \quad \varphi(x) \quad \text{subject to} \\
& \quad (A - I) \begin{pmatrix} x \\ s \end{pmatrix} = 0, \quad \ell \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u.
\end{align*}
\]

We summarize the main features of the method in this section.

2.1. Notation

Let \( g(x) \) denote the gradient of the quadratic objective \( \varphi \) so that \( g(x) = c + Hx \). For any vector \( v \), the notation \( v_S \) denotes the components of \( v \) with indices in the set \( S \). For a matrix \( M \), \( M_S \) denotes the submatrix formed by the columns of \( M \) with indices in \( S \).

2.2. Active-set method

An active set is defined at a point \((x, s)\) as the set of indices of the variables that are active, i.e., variables that lie on their bounds. Active-set methods are iterative methods that attempt to estimate the active set at the solution of the quadratic program. To estimate the optimal active set, SQIC partitions the linear constraints \( Ax - s = b \) into the form

\[
Bx_B + Sx_S + Nx_N = b,
\]

where the basis matrix \( B \) is square and nonsingular, and the matrices \( S, N \) are the remaining columns of \((A - I)\). The vectors \( x_B, x_S, x_N \) are the associated basic, superbasic, and nonbasic variables components of \((x, s)\). The set of nonbasic variables is denoted by \( N = \{\nu_1,\ldots,\nu_n\} \), while the set of basic and superbasic variables is \( B \). Given the basic set, the matrix \( A_B \) is defined as \((B \ S)\), consisting of \( n_B \) columns of \( A \) with indices in \( B \).

The nonbinding direction method uses an inertia-controlling strategy to ensure that linear systems solved during the process remain nonsingular. The method starts at a point \((x, s)\) with associated basic and nonbasic sets \( B \) and \( N \) such that the reduced gradient \( g_B = A_B^T \pi \) and the reduced Hessian \( Z_B^T H_B Z_B \) is positive definite, where the columns of \( Z_B \) form a nullspace for \( A_B \). Such a point is called a subspace minimizer with respect to \( B \). The condition that the reduced Hessian is positive definite is equivalent to the reduced KKT matrix

\[
\begin{pmatrix}
H_B & A_B^T \\
A_B & 0
\end{pmatrix}
\]

being nonsingular with \( n_B \) positive eigenvalues and \( m \) negative eigenvalues. Every successive iterate computed by the method remains a subspace minimizer thereby ensuring the nonsingularity of the KKT matrix.

Given a nonoptimal subspace minimizer \((x, s)\), some multiplier associated with a nonbasic bound, say the \( \nu_s \)-th nonbasic variable, must be nonoptimal. The method proceeds by computing a descent direction \( p \) such that such that \((x, s) + p \) remains active for all bounds in the nonbasic set \( N \) except the \( s \)-th one and also remains feasible with respect to the equalities \( Ax - s = 0 \). In other words, \( p \) satisfies

\[
p_N = \pm e_s \quad \text{and} \quad (A - I) p = 0.
\]

This direction is computed via the block-LU method (Section 2.3) or the variable-reduction method (Section 2.4).
Once the direction is computed, the **optimal step** $\alpha_\ast$ is computed as the step that minimizes $\varphi(x + \alpha p)$ as a function of $\alpha$. Since the next iterate must remain feasible, the **maximum feasible step** $\alpha_F$ is also computed to ensure that $(x, s) + \alpha p$ remains feasible with respect to the bounds, i.e.,

\[ \ell \leq \begin{pmatrix} x \\ s \end{pmatrix} + \alpha p \leq u. \]

The final nonnegative step $\alpha$ is taken as $\min\{\alpha_\ast, \alpha_F\}$.

Given the direction and step, the next iterate is defined as $(\tilde{x}, \tilde{s}) = (x, s) + \alpha p$ with updated basic and nonbasic sets. Depending on the step taken, a variable is either removed from the nonbasic set, added to the nonbasic set, or a variable in the nonbasic set is swapped with one from the basic set. In the case of $\alpha = \alpha_F$, a blocking constraining $\beta_r \in B$ is identified and second linear system involving the reduced KKT matrix is solved to determine the linear dependence of a set of constraints. If a particular component of the solution vector is below a certain tolerance, then the constraints are linearly dependent (see also 5.4).

It can be shown that $(\tilde{x}, \tilde{s})$ with its updated basic and nonbasic sets remains a subspace minimizer. The method continues until an optimal point is found or the problem is determined to be unbounded.

### 2.3. Schur complement and block LU methods

As described, the basic and nonbasic sets change by at most one variable at each iteration resulting in a different reduced KKT matrix at each iterate. Instead of reformulating the matrix (2.1) at every iteration, **SQIC** stores the initial reduced KKT matrix

\[ K = \begin{pmatrix} H_B & A_B^T \\ A_B & 0 \end{pmatrix} \]

and implements a **Schur complement method** that augments the matrix to reflect the changes to the basic set.

In addition, the Schur complement method is extended to a **block-LU method** by storing the augmented matrix in block factors

\[
\begin{pmatrix} K & V \\ V^T & D \end{pmatrix} = \begin{pmatrix} L & Y \\ Z^T & C \end{pmatrix},
\]

where $K = LU$, $LY = V$, $U^TZ = V$, and $C = D - Z^TY$ is the Schur-complement matrix. Thus, changes to $B$ are reflected by updates to the sparse matrices $Y$ and $Z$ and the Schur complement $C$. These updates involve solving linear systems with $L$ and $U$, which are performed by the third-party linear solvers.

### 2.4. Variable-reduction method

In addition to the block-LU method described above, **SQIC** can also use a variable-reduction method to store and update the matrices required to solve the quadratic problem. Unlike the block-LU method, variable-reduction keeps a sparse LU factor of a nonsingular basis matrix $B$ and a Cholesky factorization of the (positive definite) reduced Hessian matrix $Z^T H Z$.

The efficiency of variable-reduction method is linked to the size of the reduced Hessian, which is equal to the number of superbasic variables. Therefore, the variable-reduction method is used only when the number of superbasic variables is below 2000. If $n_s$ increases to above 2000, **SQIC** will switch to the block-LU method if possible.

Further details of the method and proofs are available in Gill and Wong [8], and Wong [11].
3. Installing SQIC

SQIC can be easily configured, compiled and installed with the usual ./configure, make, make install commands.

<table>
<thead>
<tr>
<th>Configure Options</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>--prefix=/some/place</td>
<td>Specify a location to install the libraries (the default is $SQIC/lib, where $SQIC is the build directory). The libraries will be installed into /some/place/lib</td>
</tr>
<tr>
<td>--with-ma57=/path/to/ma57</td>
<td>Specify the location of the HSL_MA57 libraries and module files. The configure script will search in /path/to/ma57/lib for the library and /path/to/ma57/src for the module files</td>
</tr>
<tr>
<td>--with-umfpack=/path/to/umfpack</td>
<td>Specify the location of the UMFPACK libraries. The configure script will search in /path/to/umfpack/lib for the UMFPACK libraries and in /path/to/umfpack/include for the UMFPACK header files. It will automatically search for the various UMFPACK libraries (camd, cholmod, colamd, ccolamd) and their header files (see the next entry)</td>
</tr>
<tr>
<td>--with-camd=yes/no, --with-cholmod=yes/no, --with-colamd=yes/no, --with-ccolamd=yes/no</td>
<td>Use these flags to enable or disable the various UMFPACK libraries</td>
</tr>
<tr>
<td>--with-superlu=/path/to/superlu</td>
<td>Specify the location of the SUPERLU libraries. The configure script will search in /path/to/superlu/lib for the SUPERLU library</td>
</tr>
<tr>
<td>--with-blas</td>
<td>Specify a third-party BLAS library. For example, on MacOSX, the user may specify --with-blas=-framework Accelerate; for Linux, --with-blas=-lblas</td>
</tr>
<tr>
<td>--with-debug</td>
<td>Compile SQIC with debug flags -g</td>
</tr>
<tr>
<td>--with-32, --with-64, --with-128</td>
<td>Compile SQIC with normal precision (32-bit ints, 64-bit reals), double precision (64-bit ints, 64-bit reals), quad precision (64-bit ints, 128-bit reals)</td>
</tr>
</tbody>
</table>

The user may also choose to build SQIC in a separate directory from the source files. To do this, create a directory where you want SQIC to be built and move into this directory, e.g.,

```bash
gs mkdir sqic-build
> cd sqic-build
```

Run the configure script in the SQIC directory with the command

```bash
> /path/to/sqic/configure
```

To compile the code,

```bash
> make
```

and to install,

```bash
> make install
```

The user can choose to compile only certain parts of SQIC.

```bash
> make all  % compile everything
> make library  % compile only the libraries
> make examples  % compile the libraries and examples;
                % note that this will also install the libraries
```
4. Using SQIC

The SQIC package is accessed via a derived type `qpProb` in the Fortran module SQIC, which must be 'USE'd in the user-written main program. The details and usage of the type `qpProb` are discussed in the following sections.

4.1. Subroutines associated with SQIC

The SQIC is accessed via the derived type `qpProb` and its associated subroutines:

- `begin` (Section 4.2) must be called before any calls to other SQIC routines.
- `specs` (Section 5.2) may be called to input a Specs file (a list of run-time options).
- `setSize` (Section 4.3) sets the problem size and allocates space for the problem.
- `setConstraints` (Section 4.4) initializes and allocates space for the constraint matrix of the problem.
- `setHessian` (Section 4.5) sets the Hessian matrix of the problem.
- `solve` (Section 4.7) is the main solver.
- `end` (Section 4.8) must be called at the end of the main program to deallocate any internal workspace structures.
- `usrHx` (Section 4.9) is the (optional) user-defined subroutine that computes the matrix-vector product $Hx$ for a given vector $x$.
- `qpProb` (Section 4.6) is the derived type that is used to supply problem data to the solver and to access relevant subroutines.

4.2. Subroutine `begin`

This subroutine sets the Print and Summary files and initializes the options for the solver. `begin` must be called before calls to any other SQIC subroutines.

```fortran
    call begin ( iPrint, iSumm, myProb, options )
```

**On entry:**
- `iPrint` defines a unit number for a Print file. Typically, `iPrint = 9`. If `iPrint ≤ 0`, then no Print file is output.
- `iSumm` defines a unit number for a Summary file. Typically, `iPrint = 6`. If `iSumm ≤ 0`, then no Summary file is output.
- `myProb` is the problem object.
- `options` is the options structure of type `snOpts`. Parameters are initialized to default settings.

4.3. Subroutine `setSize`

This subroutine allocates the space needed inside the `qpProb` type. The following required components of the `qpProb` structure are initialized and allocated:

```fortran
    n, m, ncObj, hEtype(:), hs(:,), bl(:,), bu(:,), cObj(:,), x(:,), pi(:,), rc(:,)
```

Optional components are also allocated if provided:

```fortran
    nNames and Names(:)
```

After allocating the structures, the user must set them to the appropriate values.

```fortran
    call setSize ( myProb, m, n, ncObj, nNames, nnzA, options )
    call setSize ( myProb, m, n, ncObj, nnzA, options )
```
4. Using SQIC

On entry:

\( m \) is the number of linear constraints in the problem. \( m \) must be greater than 1. If no constraints exist, a dummy constraint with no lower and upper bounds can be input.

\( n \) is the number of variables in the problem.

\( n_{\text{Names}} \) is an OPTIONAL argument giving the size of the character array \( \text{Name}_s(:) \). If given, the value must be equal to \( n + m \). If it is not given, then the user does not need to provide names via the \( \text{Name}_s(:) \) component for the variables and constraints.

\( nc_{\text{Obj}} \) indicates the length of the dense linear objective term \( c_{\text{Obj}} \) (\( 0 \leq nc_{\text{Obj}} \leq n \)).

\( nnz_A \) is an integer specifying the number of nonzero elements in \( A \).

\( \text{options} \) is the options structure of type \( \text{sn}_\text{opts} \).

4.4. Subroutine setConstraints

This subroutine initializes the constraint matrix of the problem. The components of the constraint matrix

\[
A%nnz, A%\text{Ind}(:,), A%\text{Loc}(:,), A%\text{Val}(:,)
\]

\begin{verbatim}
call setConstraints ( myProb, m, n, nnzA, Errors, options )
\end{verbatim}

On entry:

\( m \) is the number of linear constraints in the problem. \( m \) must be greater than 1. If no constraints exist, a dummy constraint with no lower and upper bounds can be input.

\( n \) is the number of variables in the problem.

\( nnz_A \) is an integer specifying the number of nonzero elements in \( A \).

\( \text{options} \) is an object of type \( \text{qp}_\text{opts} \) containing the options for the problem.

On exit:

\( \text{Errors} \) is the number of errors encountered.

4.5. Subroutine setHessian

This subroutine sets the Hessian matrix of the problem. If the problem is linear, no call is necessary.

\begin{verbatim}
! For Hessian defined by a subroutine:
call setHessian ( myProb, nnH, usrHx, Errors, options )

! For sparse-by-column Hessian:
call setHessian ( myProb, nnH, neH, symmetric, Errors, options )
\end{verbatim}

If the Hessian is stored as a sparse-by-column matrix, then the user must set the components inside the \( \text{qp}_\text{Prob} \) object after calling this subroutine. See Section 4.6 for details on \( \text{qp}_\text{Prob} \).
On entry:

\( nnH \) is the number of leading nonzero columns of the QP Hessian \( (0 \leq nnH \leq n) \). If \( nnH = 0 \), then the problem is linear. If \( nnH > 0 \), the user must provide \( \text{usrHx} \) to compute the matrix-vector product \( Hx \) or store the matrix in \( H \).

\( neH \) is an integer specifying the number of nonzero elements in \( H \).

\( \text{symmetric} \) is a logical. If \( \text{symmetric} \) is true, then the user will only provide the lower-triangular part of the Hessian in sparse-by-column format. Otherwise, the user will provide the full matrix. Providing the full matrix can speed up the solve time.

\( \text{usrHx} \) is a pointer to the user-defined procedure for computing Hessian matrix-vector products if \( H \) is undefined. See Section 4.9.

On exit:

\( \text{Errors} \) is the number of errors encountered.

4.6. \textit{qpProb} type

The \textit{qpProb} type supplies problem data to the solver. In particular, information regarding the constraint matrix \( A \), the lower and upper bounds \( \ell \) and \( u \), the linear and constant terms \( c \) and \( \phi \) of the objective and the Hessian matrix \( H \) of the objective are passed to \textit{SQIC} via this structure.

The Hessian matrix can be specified in sparse-by-column format or via a user-defined procedure \( \text{usrHx} \) that computes the matrix-vector product \( Hx \) for a given vector (see Section 4.9). Note that despite being symmetric, the entire matrix must be provided when using sparse-by-column format.

The linear term of the quadratic objective can be specified in three ways: as a sparse row of \( A \), as a dense vector \( c \), or as both. When stored in \( A \), the variable \( iObj \) must be set to the row of \( A \) containing the vector. The objective row must be \textit{free} with lower and upper bounds defined as \(-\infty\) and \(+\infty\).

To initialize and set a structure of type \textit{qpProb}:

use \textit{SQIC}
...
type(qpProb) :: myProb
...
myProb%name = 'myName'
call setSize ( myProb, m, n, ncObj, nNames, Errors, options )

myProb%Names(1:nNames) = ...
myProb%cObj(1:ncObj) = ...
myProb%hs(1:n+m) = ...
myProb%bl(1:n+m) = ...
myProb%bu(1:n+m) = ...
myProb%x(1:n+m) = ...
myProb%pi(1:m) = ...
myProb%rc(1:n+m) = ...
myProb%iObj = ...
myProb%ObjAdd = ...
4. Using SQIC

call setConstraints ( myProb, m, n, nnzA, Errors, options )

myProb%A%ind(1:nnzA) = ...
myProb%A%loc(1:n+1) = ...
myProb%A%val(1:nnzA) = ...

call setHessian( myProb, nnH, usrHx, Errors, options ) ! user-defined subroutine H

call setHessian( myProb, nnH, neH, symm, Errors, options ) ! sparse-by-column H

! If H is sparse-by-column, then define H:
myProb%H%val(1:neH) = ...
myProb%H%ind(1:neH) = ...
myProb%H%loc(1:nnH+1) = ...

Type components:
Values must be defined by the user in the main program.

name is an 8-character name for the problem. A blank name may be used.
m, n are integers specifying the number of constraints in the problem used and the number of variables (i.e., the size of A is m by n).

Note that A must have at least one row. If your problem has no constraints, or only upper and lower bounds on the variables, then you must include a dummy row with sufficiently wide upper and lower bounds.

nnH is the number of leading nonzero columns of the QP Hessian (0 ≤ nnH ≤ n). If nnH = 0, then the problem is linear. If nnH > 0, the user must provide usrHx to compute the matrix-vector product Hx or store the matrix in H.

iObj says which row of A is a free row containing a sparse linear objective vector c (0 ≤ iObj ≤ m). If no such row exists, iObj = 0.

ObjAdd is the constant φ added to the objective for printing purposes. Typically, ObjAdd is zero.

nnzA is an integer specifying the number of nonzero elements in A.

A%A%ind(:,), A%A%loc(:,), A%A%val(,,:) define the nonzero elements of the constraint matrix. A%A%loc is a pointer to an integer array of size n + 1 containing indices of the start of each column in A%A%ind and A%A%val. A%A%ind is a pointer to an integer array of size A%A%nnz and A%A%val is a pointer to a real array of size A%A%nnz. It is required that A%A%loc(1) = 1 and A%A%loc(n+1) = A%A%nnz + 1.

Consider the j-th column of a matrix. Then for k = A%A%loc(j) : A%A%loc(j+1) − 1, A%A%ind(k) and A%A%val(k) contain the row index and the value of the nonzero entries in column j.

bl(n+m), bu(n+m) are pointers to real arrays of size n + m containing the lower and upper bounds on the variables. The first n entries of bl, bu, refer to the variables x. The last m entries refer to the slacks s. It is required that bl(j) ≤ bu(j) for all j.

To specify non-existent bounds, set bl(j) ≤ −infBnd or bu(j) ≥ infBnd, where infBnd is the Infinite Bound size (default value 10^20). For equality bounds, set bl(j) = bu(j).

ncObj indicates the length of the dense linear objective term cObj (0 ≤ ncObj ≤ n).

cObj(ncObj) is a pointer to a real array of size ncObj containing the linear objective vector c.

If ncObj < n, then the first ncObj elements of x correspond to the vector c.

Names(nName) is an optional parameter containing 8-character names for the variables and constraints. If nName = 1, then Names is not used. The printed solution will use generic names for the columns and row. Otherwise, nName = n + m and Names(j) should contain the 8-character name of the jth variable (j = 1 : n + m). If j = n + i, the jth variable is the ith row.
The values $\text{hEtype}(j) = 0, 1, 2, 3$ have the following meaning:

<table>
<thead>
<tr>
<th>$\text{hEtype}(j)$</th>
<th>Status in elastic mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>variable $j$ is non-elastic and cannot be infeasible</td>
</tr>
<tr>
<td>1</td>
<td>variable $j$ may violate its lower bound</td>
</tr>
<tr>
<td>2</td>
<td>variable $j$ may violate its upper bound</td>
</tr>
<tr>
<td>3</td>
<td>variable $j$ may violate either of its bounds</td>
</tr>
</tbody>
</table>

$h\text{Etype}$ need not be assigned if $\text{Elastic mode} = 0$.

$\text{hs}(n+m)$ is a pointer to an integer array of size $n + m$ sometimes containing a set of initial states for $x$.

$x(n+m)$ is a pointer to a real array of size $n + m$ sometimes containing a set of initial values for $x$.

$\text{rc}(n+m)$ is a pointer to a real array of size $n + m$. It does not need to be initialized by the user.

$\text{pi}(m)$ is a pointer to a real array of size $m$ that may contain a set of initial multiplier values for $\pi$.

$\text{usrHx}$ is a pointer to the user-defined procedure for computing Hessian matrix-vector products if $H$ is undefined. See Section 4.9.

$\text{neH}$ is an integer specifying the number of nonzero elements in $H$.

$H, H\%\text{indH}(:), H\%\text{locH}(:), H\%\text{valH}(:)$ define the nonzero elements of the Hessian matrix. Although the Hessian is symmetric, to save time, SQIC requires users to provide the entire matrix in sparse-by-column format. $\text{locH}$ is a pointer to an integer array of size $\text{nnH} + 1$ containing indices of the start of each column in $\text{indH}$ and $\text{valH}$. $\text{indH}$ is a pointer to an integer array of size $\text{neH}$ and $\text{valH}$ is a pointer to a real array of size $\text{neH}$. It is required that $\text{locH}(1) = 1$ and $\text{locH}(n + 1) = \text{neH} + 1$.

4.7. Subroutine SQIC

The SQIC solver is accessed via the $\text{solve}$ subroutine.

```fortran
    call solve ( Start, myProb, options, output, INFO )
```

**On entry:**

$\text{Start}$ is of type character*(*) specifying the type of start.

$\text{options}$ is the options structure of type $\text{snOpts}$.

**On exit:**

$\text{INFO}$ reports the status of the call to SQIC.

- 0 finished successfully
- 11 infeasible problem
- 21 unbounded objective
- 31 iteration limit reached
- 34 time limit reached
- 40 encountered numerical difficulties
- 80 memory allocation error
- 90 invalid input argument
- 140 internal error
- 150 LU solver error

$\text{nS}$ is the number of superbasics at the solution of the problem.

$\text{nInf, sInf}$ are the number of and the sum of infeasibilities if the problem is deemed infeasible.
4. Using SQIC

Obj is the final quadratic objective value at the point \( x \). Note that Obj does not contain the constant term ObjAdd or the objective row (if one exists). The final value of the objective being optimized is ObjAdd + \( x(n+iObj) \) + Obj if \( iObj > 0 \) or ObjAdd + Obj if \( iObj = 0 \).

4.8. Subroutine end

end must be called at the end of the user’s main program to deallocate any internal workspace used in SQIC.

\[
\text{call end ( myProb )}
\]

On entry:

options is the options structure of type snOpt.

4.9. Subroutine usrHx

If \( H \) is not stored as a sparse matrix, then the \text{usrHx} subroutine must be defined by the user to computes the matrix-vector product \( Hx \) for a given vector \( x \) or a specified column of \( H \).

\[
\text{subroutine usrHx ( nnH, x, Hx, jcol, State )}
\]

\[
\begin{align*}
\text{integer(ip), intent(in)} & : \text{nnH, jcol, State} \\
\text{real(rp), intent(in)} & : \text{x(nnH)} \\
\text{real(rp), intent(out)} & : \text{Hx(nnH)}
\end{align*}
\]

On entry:

nnH is the number of leading nonzero columns of the QP Hessian.

x is a real array of size \( \text{nnH} \) that is multiplied by the Hessian matrix.

jcol indicates the column of \( H \) to return. If \( jcol = 0 \), then the user must compute the matrix-vector product \( Hx \) with the given \( x \). If \( jcol > 0 \), then the user must return the \( jcol \)-th column of \( H \). In this case, \( x \) may be undefined.

State indicates the status of the call to \text{usrHx}. If Status = 0, there is nothing special about the call. If Status = 1, then SQIC is calling the subroutine for the first time. If Status \geq 2, then SQIC is calling the subroutine for the last time.

On exit:

Hx contains the result of the matrix-vector product \( Hx \) or if \( jcol > 0 \), the \( jcol \)-th column of \( H \).
5. Optional parameters

The performance of each SQIC interface is controlled by a number of parameters or “options”. Each option has a default value that should be appropriate for most problems. Other values may be specified by calling the subroutine specs to read a Specs file.

5.1. The SPECS file

The Specs file contains a list of options and values in the following general form:

Begin options
  QPsolver  variable
  BlockLU   LUSOL
  Solution  Yes
End options

We call such data a Specs file because it specifies various options. The file starts with the keyword Begin and ends with End. The file is in free format. Each line specifies a single option, using one or more items as follows:

1. A keyword (required for all options).
2. A phrase (one or more words) that qualifies the keyword (only for some options).
3. A number that specifies an integer or real value (only for some options). Such numbers may be up to 16 contiguous characters in Fortran 77’s I, F, E or D formats, terminated by a space or new line.

The items may be entered in upper or lower case or a mixture of both. Some of the keywords have synonyms, and certain abbreviations are allowed, as long as there is no ambiguity. Blank lines and comments may be used to improve readability. A comment begins with an asterisk (*) anywhere on a line. All subsequent characters on the line are ignored.

The Begin line is echoed to the Summary file.

5.2. Subroutine specs

specs may be called to input a Specs file. The options available are listed in Section 5.

    call specs ( iSpecs, options, iExit )

On entry:

iSpecs is a unit number for the Specs file. Typically, iSpecs = 4.
options is the options structure of type snOpts.

On exit:

options will be set based on the specifications file.
iExit reports the result of calling specs. Here is a summary of possible values.

    Finished successfully
    101 Specs file read.

    Errors while reading Specs file
    131 No Specs file specified (iSpecs ≤ 0 or iSpecs > 99).
    132 End-of-file encountered while looking for Specs file. specs encountered end-of-file or Endrun before finding Begin (see Section 5.3). The Specs file may not be properly assigned.
5. Optional parameters

133 End-of-file encountered before finding \texttt{End}. Lines containing \texttt{Skip} or \texttt{Endrun} may imply that all options should be ignored.

134 \texttt{Endrun} found before any valid sets of options.

> 134 There were $i = \text{INFO} - 134$ errors while reading the Specs file.

5.3. Multiple sets of options in the Specs file

The keyword \texttt{Skip} allows you to collect several sets of options within a single Specs file. In the following example, only the second set of options will be input.

\begin{verbatim}
Skip Begin options
  BlockLU UMFPACK
End options

Begin options 2
  BlockLU MA57
End options 2
\end{verbatim}

The keyword \texttt{Endrun} prevents subroutine \texttt{specs} from reading past that point in the Specs file while looking for \texttt{Begin}.

5.4. Description of Optional Parameters

The following shows all valid keywords specific to the SQIC interface and their default values.

BEGIN checklist of SPECS file parameters and their default values

\begin{verbatim}
BlockLU LUSOL * LUSOL
End of SPECS file checklist
\end{verbatim}

The following is an alphabetical list of the options specific to SQIC that may appear in the Specs file, and a description of their effect. For more keywords, see the SNOPT User's Guide [7].

\begin{verbatim}
BlockLU LUSOL 1
BlockLU MA57 2
BlockLU UMFPACK 3
\end{verbatim}

This option specifies the linear solver to be used. If a specified solver is unavailable, then SQIC uses the default LUSOL. The solver must already be compiled and linked. The solver is specified by either its name or the number associated with the solver as listed above.

\begin{verbatim}
Debug \ i \ Default = 0
\end{verbatim}

If \texttt{debug} is greater than 0, then consistency checks are done and any inconsistent results are printed. If \texttt{debug} is greater than 10, then the accuracy of the solutions to linear systems is checked and the norms are printed. These checks generally will slow down performance.

\begin{verbatim}
Inertia \ i \ Default = 1000000
\end{verbatim}

This option specifies the number of iterations to perform an explicit inertia check of the KKT matrix. In general, this is a very expensive operation so by default it is never done.

\begin{verbatim}
Independence tolerance \ \tau \ Default = 5.0d - 9
\end{verbatim}

This option sets the tolerance used in the linear dependence test

$$u_{\beta_r} \leq \tau p_{\beta_r}.$$
where $\beta_r$ is a blocking constraint, and $p$ and $u$ are solutions of the first and second linear systems solved during an iteration of the algorithm. In general, the user should not modify this option.

<table>
<thead>
<tr>
<th>QPmode</th>
<th></th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQOPT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This specifies which QP solver to use initially. For **general**, **SQIC** will start in variable-reduction mode if the number of superbasics is below the maximum allowed. If the number of superbasics is above the maximum, **SQIC** will use enter block-LU mode. This option, in combination with the **Switch** option, is useful if the user wants to force **SQIC** to use only one method.

<table>
<thead>
<tr>
<th>Scale option</th>
<th>$i$</th>
<th>Default = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scaling iterations</td>
<td>$j$</td>
<td>Default = 10</td>
</tr>
</tbody>
</table>

There are 2 available scaling options.

- **i**
  - **Meaning**
  - 0: No scaling.
  - 1: A iterative scaling scheme is applied to $H$ and $A$ based on the estimated condition number of the KKT matrix formed by $H$ and $A$.

Scaling iterations is the maximum number of iterations performed when the scaling option is set to 1.

<table>
<thead>
<tr>
<th>Schur tolerance</th>
<th>$\tau$</th>
<th>Default = $10^8$</th>
</tr>
</thead>
</table>

This is the upper limit of the condition number of the Schur complement matrix. If the estimated condition number is larger than $\tau$, then the block LU factors are recomputed and reset.

<table>
<thead>
<tr>
<th>Switch</th>
<th></th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If **Switch** is set to **YES**, then **SQIC** is allowed to switch between the variable-reduction method and the block-LU method for solving a general quadratic problem. Switching is determined by the number of superbasic variables. Otherwise, **SQIC** will not switch to a different mode and may terminate early.

<table>
<thead>
<tr>
<th>Time</th>
<th>$r$</th>
<th>Default = $10^8$</th>
</tr>
</thead>
</table>

Set a time limit in seconds. By default, the limit is set at $10^8$ seconds, which is essentially no limit.

<table>
<thead>
<tr>
<th>Vertex</th>
<th></th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If **vertex** is set to **YES**, then **SQIC** is forced to start the algorithm at a vertex.
6. Interfaces

6.1. Matlab interface

Coming soon.

6.2. C interface

An interface to the package written in C is included in the SQIC package. A typical invocation of the C version of SQIC is

```c
char *probName = ...;
char *prtFile = ...;
*sumFile = ...;
*spcFile = ...;

int m = ...;
int n = ...;
int nnH = ...;
int ncObj = ...;
int nnzA = ...;

int locA[n+1], indA[nnzA], locH[nnH+1], indH[nnH];
double infBnd, zero;
double ObjAdd, sInf, Obj;
double bl[n+m], bu[n+m], x[n+m], pi[m], rc[n+m];
double cObj[1], hEtype[n+m], hs[n+m];
double valA[nnzA], valH[nnH];

...

begin ( prtFile, sumFile );
specs ( spcFile, &INFO );
...

/**< First call passes the user-defined subroutine usrHx */
 sqic ( &iStart, probName, &m, &n, &nnH, &iObj, &ObjAdd,
 &nnzA, indA, locA, valA, bl, bu, &ncObj, cObj,
 hEtype, hs, x, pi, rc, usrHx,
 &INFO, &nS, &nInf, &sInf, &Obj );

/**< Second call passes the sparse Hessian */
 sqic_h ( &iStart, probName, &m, &n, &nnH, &iObj, &ObjAdd,
 &nnzA, indA, locA, valA, bl, bu, &ncObj, cObj,
 hEtype, hs, x, pi, rc,
 &nnH, indH, locH, valH,
 &INFO, &nS, &nInf, &sInf, &Obj );
end();
```

Remember that C uses zero-based indexing.

6.2.1. Subroutine begin

begin ( char *prtFile, char *sumFile );
On entry:

- **prtFile** is a character string containing the name of the print file.
- **sumFile** is a character string containing the name of the summary file.

### 6.2.2. Subroutine specs

```c
specs ( char *spcFile, int *INFO );
```

On entry:

- **spcFile** is a character string containing the name of the specs file.

On exit:

- **INFO** is an integer returning the result of the call to the specs subroutine.

### 6.2.3. Subroutine sqic

```c
sqic ( int *iStart, char *probName,
      int *m, int *n, int *nnH, int *iObj, double *ObjAdd,
      int *nnzA, int indA[], int locA[], double valA[],
      double bl[], double bu[], int *ncObj, int cObj[],
      int hEtype[], int hs[], double x[],
      double pi[], double rc[], void usrHx,
      int *INFO, int *nS, int *nInf, double *sInf, double *Obj );
```

```c
sqic_h ( int *iStart, char *probName,
         int *m, int *n, int *nnH, int *iObj, double *ObjAdd,
         int *nnzA, int indA[], int locA[], double valA[],
         double bl[], double bu[], int *ncObj, int cObj[],
         int hEtype[], int hs[], double x[],
         double pi[], double rc[], void usrHx,
         int *nnH, int indH[], int locH[], double valH[],
         int *INFO, int *nS, int *nInf, double *sInf, double *Obj );
```

On entry:

- **iStart** is an integer specifying the Start type. If **iStart** = 0, then **SQIC** uses a Cold start. If **iStart** = 1, then a Warm start is used.
- **m**, **n** are integers specifying the number of constraints in the problem used and the number of variables (i.e., the size of A is **m** by **n**).
  
  *Note that A must have at least one row.* If your problem has no constraints, or only upper and lower bounds on the variables, then you must include a dummy row with sufficiently wide upper and lower bounds.
- **nnH** is the number of leading nonzero columns of the QP Hessian (0 ≤ nnH ≤ n). If nnH = 0, then the problem is linear. If nnH > 0, the user must provide **usrHx** to compute the matrix-vector product Hx or provided in a sparse-by-column format.
- **iObj** says which row of A is a free row containing a sparse linear objective vector c (0 ≤ iObj ≤ m). If no such row exists, iObj = 0.
- **ObjAdd** is the constant φ added to the objective for printing purposes. Typically, ObjAdd is zero.
- **nnzA** is an integer specifying the number of nonzero elements in A.
- **indA[]**, **locA[]**, **valA[]** define the nonzero elements of the constraint matrix. **locA** is an integer array of size n + 1 containing indices of the start of each column in **indA** and **valA**.
is a pointer to an integer array of size nnzA and valA is a pointer to a real array of size nnzA. It is required that locA[0] = 1 and locA[n] = nnzA + 1.

Consider the j-th column of a matrix. Then for \( k = \text{locA}[j] : \text{locA}[j + 1] - 1 \), indA(k) and valA(k) contain the row index and the value of the nonzero entries in column j.

bl[n+m], bu[n+m] are pointers to real arrays of size n + m containing the lower and upper bounds on the variables. The first n entries of bl, bu, refer to the variables x. The last m entries refer to the slacks s. It is required that \( bl[j] \leq bu[j] \) for all j.

To specify non-existent bounds, set \( bl[j] \leq -\text{infBnd} \) or \( bu[j] \geq \text{infBnd} \), where \( \text{infBnd} \) is the Infune Bound size (default value \( 10^{20} \)). For equality bounds, set \( bl[j] = bu[j] \).

\( \text{ncObj} \) indicates the length of the dense linear objective term cObj (0 ≤ \( \text{ncObj} \) ≤ n).

\( \text{cObj}[\text{ncObj}] \) is a pointer to a real array of size \( \text{ncObj} \) containing the linear objective vector c. If \( \text{ncObj} < n \), then the first \( \text{ncObj} \) elements of x correspond to the vector c.

\( \text{hEtype}[n+m] \) sometimes defines which variables are to be treated as being elastic in elastic mode.

The values \( \text{hEtype}[j] = 0, 1, 2, 3 \) have the following meaning:

<table>
<thead>
<tr>
<th>( \text{hEtype}[j] )</th>
<th>Status in elastic mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>variable j is non-elastic and cannot be infeasible</td>
</tr>
<tr>
<td>1</td>
<td>variable j may violate its lower bound</td>
</tr>
<tr>
<td>2</td>
<td>variable j may violate its upper bound</td>
</tr>
<tr>
<td>3</td>
<td>variable j may violate either of its bounds</td>
</tr>
</tbody>
</table>

\( \text{hEtype} \) need not be assigned if Elastic mode = 0.

\( \text{hs}[n+m] \) is a pointer to an integer array of size n + m sometimes containing a set of initial states for x.

\( \text{x}[n+m] \) is a pointer to a real array of size n + m sometimes containing a set of initial values for x.

\( \text{rc}[n+m] \) is a pointer to a real array of size n + m. It does not need to initialized by the user.

\( \text{pi}[m] \) is a pointer to a real array of size m that may contain a set of initial multiplier values for \( \pi \).

\( \text{usrHx} \) is a function to the user-defined procedure for computing Hessian matrix-vector products. See Section 6.2.5.

\( \text{neH} \) is an integer specifying the number of nonzero elements in \( H \).

\( \text{indH}[], \text{locH}[], \text{valH}[] \) define the nonzero elements of the Hessian matrix. Although the Hessian is symmetric, to save time, SQIC requires users to provide the entire matrix in sparse-by-column format. \( \text{locH} \) is a pointer to an integer array of size \( \text{neH} + 1 \) containing indices of the start of each column in \( \text{indH} \) and \( \text{valH} \). \( \text{indH} \) is a pointer to an integer array of size \( \text{neH} \) and \( \text{valH} \) is a pointer to a real array of size \( \text{neH} \). It is required that \( \text{locH}[0] = 1 \) and \( \text{locH}[n + 1] = \text{neH} + 1 \).

On exit:
\( \text{INFO} \) is an integer returning the result of the call to the specs subroutine.

6.2.4. Subroutine end

\text{end () ;}

6.2.5. Subroutine \text{usrHx}

If \( H \) is not stored as a sparse matrix, then the \text{usrHx} subroutine must be defined by the user to computes the matrix-vector product \( Hx \) for a given vector \( x \) or a specified column of \( H \).

\text{void usrHx ( int *nnH, double x[], double Hx[], int *jcol, int *State )}
On entry:

\( n_{\text{H}} \) is the number of leading nonzero columns of the QP Hessian.

\( x \) is a double array of size \( n_{\text{H}} \) that is multiplied by the Hessian matrix.

\( j_{\text{col}} \) indicates the column of \( H \) to return. If \( j_{\text{col}} = 0 \), then the user must compute the matrix-vector product \( Hx \) with the given \( x \). If \( j_{\text{col}} > 0 \), then the user must return the \( j_{\text{col}} \)-th column of \( H \). In this case, \( x \) may be undefined.

\textbf{State} indicates the status of the call to \texttt{usrHx}. If \texttt{Status} = 0, there is nothing special about the call. If \texttt{Status} = 1, then \texttt{SQIC} is calling the subroutine for the first time. If \texttt{Status} \geq 2, then \texttt{SQIC} is calling the subroutine for the last time.

On exit:

\( Hx \) contains the result of the matrix-vector product \( Hx \) or if \( j_{\text{col}} > 0 \), the \( j_{\text{col}} \)-th column of \( H \).

6.3. CUTEst interface
References


