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.

*Partially supported by
†pgill@ucsd.edu (http://www.ccom.ucsd.edu/~peg)
‡elwong@ucsd.edu (http://www.ccom.ucsd.edu/~elwong)

Contents

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

$$\text{QP} \qquad \begin{array}{l} \underset{x}{\text{minimize}} \quad \varphi(x) \\ \text{subject to} \quad \ell \leq \begin{pmatrix} x \\ Ax \end{pmatrix} \leq u, \end{array}$$

where $x \in \mathbb{R}^n$, $\varphi(x)$ is a linear or quadratic objective function, ℓ 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

$$\varphi(x) = \phi + c^T x + \tfrac{1}{2} x^T H x,$$

where ϕ is a scalar constant, c is the constant linear objective vector, and H is the symmetric Hessian of the quadratic objective φ. 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

$$\underset{x,s}{\text{minimize}} \;\; \varphi(x) \quad \text{subject to} \quad Ax - s = 0, \quad \ell \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u.$$

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

```
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:

$$\underset{x,s}{\text{minimize}} \;\; \varphi(x) \quad \text{subject to} \quad \begin{pmatrix} A & -I \end{pmatrix} \begin{pmatrix} x \\ s \end{pmatrix} = 0, \quad \ell \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u.$$

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

2.1. Notation

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

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 $\begin{pmatrix} A & -I \end{pmatrix}$. 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 $\mathcal{N} = \{\nu_1, \ldots, \nu_{n_N}\}$, while the set of basic and superbasic variables is $\mathcal{B}$. Given the basic set, the matrix A_B is defined as $\begin{pmatrix} B & S \end{pmatrix}$, consisting of n_B columns of A with indices in $\mathcal{B}$.

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 $\mathcal{B}$ and $\mathcal{N}$ 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 $\mathcal{B}$. The condition that the reduced Hessian is positive definite is equivalent to the reduced KKT matrix

$$\begin{pmatrix} H_B & A_B^T \\ A_B & \end{pmatrix} \tag{2.1}$$

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 ν_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 $\mathcal{N}$ except the s-th one and also remains feasible with respect to the equalities $Ax - s = 0$. In other words, p satisfies

$$p_N = \pm e_s \;\text{ and }\; \begin{pmatrix} A & -I \end{pmatrix} p = 0.$$

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* α_* is computed as the step that minimizes $\varphi(x+\alpha p)$ as a function of α. Since the next iterate must remain feasible, the *maximum feasible step* α_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 α is taken as $\min\{\alpha_*, \alpha_F\}$.

Given the direction and step, the next iterate is defined as $(\bar{x},\bar{s}) = (x,s)+\alpha p$ 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 constraing $\beta_r \in \mathcal{B}$ 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 $(\bar{x},\bar{s})$ 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

$$K = \begin{pmatrix} H_B & A_B^T \\ A_B & \end{pmatrix} \tag{2.2}$$

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

$$\begin{pmatrix} K & V \\ V^T & D \end{pmatrix} = \begin{pmatrix} L & \\ Z^T & I \end{pmatrix} \begin{pmatrix} U & Y \\ & C \end{pmatrix},$$

where $K = LU$, $LY = V$, $U^TZ = V$, and $C = D - Z^TY$ is the Schur-complement matrix. Thus, changes to $\mathcal{B}$ 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^THZ.

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.

Configure Options	Value
--prefix=/some/place	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
--with-ma57=/path/to/ma57	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
--with-umfpack=/path/to/umfpack	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)
--with-camd=yes/no, --with-cholmod=yes/no, --with-colamd=yes/no, --with-ccolamd=yes/no	Use these flags to enable or disable the various UMFPACK libraries
--with-superlu=/path/to/superlu	Specify the location of the SUPERLU libraries. The configure script will search in /path/to/superlu/lib for the SUPERLU library
--with-blas	Specify a third-party BLAS library. For example, on MacOSX, the user may specify --with-blas="-framework Accelerate"; for Linux, --with-blas="-lblas"
--with-debug	Compile SQIC with debug flags -g
--with-32, --with-64, --with-128	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)

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.,

```
>> mkdir sqic-build
>> cd  sqic-build
```

Run the configure script in the SQIC directory with the command

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

To compile the code,

```
>> make
```

and to install,

```
>> make install
```

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

```
>> 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 interal 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.

```
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:

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

Optional components are also allocated if provided:

nNames and Names(:),

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

```
call setSize ( myProb, m, n, ncObj, nNames, nnzA, options )
call setSize ( myProb, m, n, ncObj, nnzA, options )
```

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.

`nNames` is an OPTIONAL argument giving the size of the character array `Names(:)`. 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 `Names(:)` component for the variables and constraints.

`ncObj` indicates the length of the dense linear objective term `cObj` ($0 \leq$ `ncObj` $\leq$ `n`).

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

`options` is the options structure of type `snOpts`.

4.4. Subroutine `setConstraints`

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

`A%nnz`, `A%ind(:)`, `A%loc(:)`, `A%val(:)`.

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

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.

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

`options` is an object of type `qpOpts` containing the options for the problem.

On exit:

`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.

```
! 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 )
```

If the Hessian is stored as a sparse-by-column matrix, then the user must set the components inside the `qpProb` object after calling this subroutine. See Section 4.6 for details on `qpProb`.

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 `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`.

`symmetric` is a logical. If `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.

`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:

`Errors` is the number of errors encountered.

4.6. `qpProb` type

The `qpProb` type supplies problem data to the solver. In particular, information regarding the constraint matrix A, the lower and upper bounds ℓ and u, the linear and constant terms c and ϕ of the objective and the Hessian matrix H of the objective are passed to `SQIC` via this structure.

The Hessian matrix can be specified in sparse-by-column format or via a user-defined procedure `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 *free* with lower and upper bounds defined as $-\infty$ and $+\infty$.

To initialize and set a structure of type `qpProb`:

```
use 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          = ...
```

```
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 specifiying 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 ($0 \leq$ `nnH` $\leq$ `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 ($0 \leq$ `iObj` $\leq$ `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%loc(:),A%val(:)` define the nonzero elements of the constraint matrix. `A%loc` is a pointer to an integer array of size `n` $+ 1$ containing indices of the start of each column in `A%ind` and `A%val`. `A%ind` is a pointer to an integer array of size `A%nnz` and `A%val` is a pointer to a real array of size `A%nnz`. It is required that `A%loc`$(1) = 1$ and `A%loc`$(n+1) =$ `A%nnz` $+ 1$.

Consider the j-th column of a matrix. Then for $k =$ `A%loc`(j) : `A%loc`$(j+1)-1$, `A%ind`(k) and `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`$(j) \leq$ `bu`(j) for all j.

To specify non-existent bounds, set `bl`$(j) \leq -$`infBnd` or `bu`$(j) \geq$ `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` ($0 \leq$ `ncObj` $\leq$ `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.

`hEtype(n+m)` sometimes defines which variables are to be treated as being elastic in elastic mode. The values `hEtype`$(j) = 0, 1, 2, 3$ have the following meaning:

`hEtype`(j)	Status in elastic mode
0	variable j is non-elastic and cannot be infeasible
1	variable j may violate its lower bound
2	variable j may violate its upper bound
3	variable j may violate either of its bounds

`hEtype` need not be assigned if `Elastic mode` $= 0$.

`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`.

`rc(n+m)` is a pointer to a real array of size `n` + `m`. It does not need to initialized by the user.

`pi(m)` is a pointer to a real array of size `m` that may contain a set of initial multiplier values for π.

`usrHx` is a pointer to the user-defined procedure for computing Hessian matrix-vector products if `H` is undefined. See Section 4.9.

`neH` is an integer specifying the number of nonzero elements in `H`.

`H,H%indH(:),H%locH(:),H%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. `locH` is a pointer to an integer array of size `nnH` + 1 containing indices of the start of each column in `indH` and `valH`. `indH` is a pointer to an integer array of size `neH` and `valH` is a pointer to a real array of size `neH`. It is required that `locH`$(1) = 1$ and `locH`$(\texttt{n} + 1) = \texttt{neH} + 1$.

4.7. Subroutine `SQIC`

The `SQIC` solver is accessed via the `solve` subroutine.

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

On entry:

`Start` is of type character*(*) specifiying the type of start.

`options` is the options structure of type `snOpts`.

On exit:

`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

`nS` is the number of superbasics at the solution of the problem.

`nInf, sInf` are the number of and the sum of infeasibilities if the problem is deemed infeasible.

`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 interal workspace used in `SQIC`.

```
call end ( myProb )
```

On entry:

`options` is the options structure of type `snOpts`.

4.9. Subroutine `usrHx`

If H is not stored as a sparse matrix, then the `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.

```
subroutine usrHx ( nnH, x, Hx, jcol, State )
  integer(ip), intent(in)  :: nnH, jcol, State
  real(rp),    intent(in)  :: x(nnH)
  real(rp),    intent(out) :: Hx(nnH)
```

On entry:

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

`x` is a real array of size `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 `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` ≥ 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.

133	End-of-file encountered before finding `End`. Lines containing `Skip` or `Endrun` may imply that all options should be ignored.
134	`Endrun` found before any valid sets of options.
> 134	There were $i =$ `INFO` $- 134$ errors while reading the Specs file.

5.3. Multiple sets of options in the Specs file

The keyword `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.

```
Skip Begin options
   BlockLU   UMFPACK
End options

Begin options 2
   BlockLU   MA57
End options 2
```

The keyword `Endrun` prevents subroutine `specs` from reading past that point in the Specs file while looking for `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
  BlockLU                   LUSOL     * LUSOL
End of SPECS file checklist
```

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].

`BlockLU`	`LUSOL`	`1`	Default
`BlockLU`	`MA57`	`2`	
`BlockLU`	`UMFPACK`	`3`	

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.

`Debug` i Default $= 0$

If `debug` is greater than 0, then consistency checks are done and any inconsistent results are printed. If `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.

`Inertia` i Default $= 1000000$

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.

`Independence tolerance` τ Default $= 5.0d - 9$

This option sets the tolerance used in the linear dependence test

$$u_{\beta_r} \leq \tau p_{\beta_r},$$

where β_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.

`QPmode`	`General`	Default
`QPmode`	`Block`	
`QPmode`	`Variable`	
`QPmode`	`SQOPT`	

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.

`Scale option`	i	Default = 1
`Scaling iterations`	j	Default = 10

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.

`Schur tolerance`	τ	Default = 10^8

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

`Switch`	`Yes`	Default
`Switch`	`No`	

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.

`Time`	r	Default = 10^8

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

`Vertex`	`No`	Default
`Vertex`	`Yes`	

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

```
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`

```
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`

```
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 );

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 specifiying 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 ($0 \leq$ `nnH` $\leq$ `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 ($0 \leq$ `iObj` $\leq$ `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`. `indA`

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 $k =$ `locA`$[j] :$ `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`$[j] \leq -$`infBnd` or `bu`$[j] \geq$ `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` ($0 \leq$ `ncObj` $\leq$ `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.

`hEtype[n+m]` sometimes defines which variables are to be treated as being elastic in elastic mode.

The values `hEtype`$[j] = 0, 1, 2, 3$ have the following meaning:

`hEtype`$[j]$	Status in elastic mode
0	variable j is non-elastic and cannot be infeasible
1	variable j may violate its lower bound
2	variable j may violate its upper bound
3	variable j may violate either of its bounds

`hEtype` need not be assigned if `Elastic mode` $= 0$.

`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`.

`rc[n+m]` is a pointer to a real array of size `n` $+$ `m`. It does not need to initialized by the user.

`pi[m]` is a pointer to a real array of size `m` that may contain a set of initial multiplier values for π.

`usrHx` is a function to the user-defined procedure for computing Hessian matrix-vector products. See Section 6.2.5.

`neH` is an integer specifying the number of nonzero elements in `H`.

`indH[],locH[],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. `locH` is a pointer to an integer array of size `nnH` $+ 1$ containing indices of the start of each column in `indH` and `valH`. `indH` is a pointer to an integer array of size `neH` and `valH` is a pointer to a real array of size `neH`. It is required that `locH`$[0] = 1$ and `locH`$[$`n` $+ 1] =$ `neH` $+ 1$.

On exit:

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

6.2.4. Subroutine `end`

```
end ();
```

6.2.5. Subroutine `usrHx`

If H is not stored as a sparse matrix, then the `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.

```
void usrHx ( int *nnH, double x[], double Hx[], int *jcol, int *State )
```

On entry:

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

`x` is a double array of size `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 `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` ≥ 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.

6.3. CUTEst interface

References

[1] T. A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Software*, 30(2):196–199, 2004. 4

[2] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Software*, 30(2):167–195, 2004. 4

[3] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse *LU* factorization. *SIAM J. Matrix Anal. Appl.*, 18(1):140–158, 1997. 4

[4] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Software*, 25(1):1–20, 1999. 4

[5] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.*, 20(3):720–755 (electronic), 1999. 4

[6] I. S. Duff. MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Software*, 30(2):118–144, 2004. 4

[7] P. E. Gill, W. Murray, and M. A. Saunders. User's guide for SNOPT Version 7: Software for large-scale nonlinear programming. Numerical Analysis Report 06-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 2006. 15

[8] P. E. Gill and E. Wong. Methods for convex and general quadratic programming. Center for Computational Mathematics Report CCoM 13-1, University of California, San Diego, La Jolla, CA, 2013. 5, 6

[9] J. D. Hogg and J. A. Scott. HSL_MA97: a bit-compatible multifrontal code for sparse symmetric systems. Technical report, Rutherford Appleton Laboratory, Oxon, UK, 2011. 4

[10] HSL. A collection of Fortran codes for large-scale scientific computation. http://www.hsl.rl.ac.uk, 2013. 4

[11] E. Wong. *Active-Set Methods for Quadratic Programming*. PhD thesis, Department of Mathematics, University of California San Diego, La Jolla, CA, 2011. 5, 6