User manual for filterSQP [*][†]

Roger Fletcher and Sven Leyffer [‡]
University of Dundee

April 1998
Version 1, June 1998
Updated, March 1999

Abstract

This paper describes a software package for the solution of Nonlinear Programming (NLP) problems. The package implements a Sequential Quadratic Programming solver with a "filter" to promote global convergence. The solver runs with a dense or a sparse linear algebra package and a robust QP solver.

Key words: Nonlinear Programming, Sequential Quadratic Programming.

1 Problem Description

The software package described in this note solves Nonlinear Programming (NLP) problems of the following form

$$(P)\left\{\begin{array}{ll} \min\limits_{x} & f(\mathbf{x}) \\ \text{subject to} & \mathbf{l}_x \leq \mathbf{x} \leq \mathbf{u}_x \\ & \mathbf{l}_c \leq \mathbf{c}(\mathbf{x}) \leq \mathbf{u}_c \end{array}\right.$$

Any linear constraints are a subset of the nonlinear constraints ($\mathbf{c}(\mathbf{x})$) and the solver takes advantage of the linear structure. The solver assumes that the simple bounds are stored first in the vectors (`blo, bup`). The linear and nonlinear constraints, however, can be mixed and appear in any order.

Note that equality constraints are included in the above formulation by setting $l_i = u_i$ for the relevant constraint. Likewise it is possible to include one-sided constraints by setting l_i to $-\infty$ or u_i to ∞, depending on which bound is required.

[*]This work was supported by EPSRC grant number GR/K51204.

[†]University of Dundee Numerical Analysis Report NA/181

[‡]University of Dundee, Department of Mathematics, Dundee, DD1 4HN, Scotland, U.K., `sleyffer@mcs.dundee.ac.uk`, `fletcher@mcs.dundee.ac.uk`

2 The Algorithm

The package implements a Sequential Quadratic Programming (SQP) trust region algorithm with a "filter" to promote global convergence. The filter is a list of pairs $(f^{(l)}, h^{(l)})$ of objective values $f^{(l)} = f(\mathbf{x}^{(l)})$ and norms of constraint violations $h^{(l)} = h(\mathbf{c}(\mathbf{x}^{(l)}))$. A new step is accepted whenever it improves the objective or the constraints compared to the filter. Otherwise the step is rejected.

Starting with $\mathbf{x}^{(k)}$, a quadratic approximation to (P) is solved within a *trust–region* defined by $\|\mathbf{d}\|_\infty \le \rho$

$$(QP) \left\{ \begin{array}{ll} \underset{d}{\min} & \frac{1}{2}\mathbf{d}^T\mathbf{W}^{(k)}\mathbf{d} + \mathbf{d}^T\mathbf{g}^{(k)} \\ \text{subject to} & \mathbf{l}_x \le \mathbf{x}^{(k)} + \mathbf{d} \le \mathbf{u}_x \\ & \mathbf{l}_c \le \mathbf{A}^{(k)^T}\mathbf{d} + \mathbf{c}^{(k)} \le \mathbf{u}_c \\ & \|\mathbf{d}\|_\infty \le \rho \end{array} \right.$$

which produces a trial step $\mathbf{d}^{(k)}$. Here $\mathbf{g}^{(k)} = \nabla f(\mathbf{x}^{(k)})$, $\mathbf{A}^{(k)} = \nabla \mathbf{c}^T(\mathbf{x}^{(k)})$
and $\mathbf{W}^{(k)} = \nabla^2 \mathcal{L}(\mathbf{x}^{(k)}, \boldsymbol{\lambda}^{(k)})$ is the Hessian of the Lagrangian $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \boldsymbol{\lambda}^T\mathbf{c}(\mathbf{x})$.

The trust region radius ρ is changed adaptively by the algorithm. If a step is rejected then the quadratic model (QP) of (P) is judged to be poor and the trust region is reduced by setting $\rho = \rho/2$. If a step is accepted on the other hand, then the trust region radius is increased by setting $\rho = \rho * 2$ The initial trust region radius is usually not critical to the success of the algorithm. For a more detailed description of the SQP algorithm see [3].

The QP problems are solved using the robust QP solver `bqpd`. `bqpd` is a null–space active set method that builds up a factorization of the reduced Hessian, that is the projection of the Hessian onto the null–space of constraints that are currently regarded as being active. The dimension of the reduced Hessian is closely related to the number of degrees of freedom of (P), and the user must set an upper bound on this number of degrees of freedom in `kmax`. Note that `kmax = n` will always be sufficient, but increases the storage requirement which is $\mathcal{O}(\texttt{kmax}^2)$.

2.1 Termination Criteria

The algorithm terminates when it has found a Kuhn–Tucker point or no further progress appears possible (this allows termination at solutions which are not Kuhn–Tucker points). The triple $(\mathbf{x}^*, \boldsymbol{\nu}^*\boldsymbol{\lambda}^*)$ is a Kuhn–Tucker point of (P) if the following conditions hold

$$\mathbf{g}^* - \boldsymbol{\nu}^* - \mathbf{A}^*\boldsymbol{\lambda}^* = \mathbf{0} \tag{1}$$

$$\left. \begin{array}{c} \mathbf{l}_x \le \mathbf{x}^* \le \mathbf{u}_x \\ \mathbf{l}_c \le \mathbf{c}^* \le \mathbf{u}_c \end{array} \right\} \tag{2}$$

$$\left.\begin{array}{rcl} x_i^* = l_{x_i} & \Rightarrow & \nu_i^* \geq 0 \\ x_i^* = u_{x_i} & \Rightarrow & \nu_i^* \leq 0 \\ l_{x_i} < x_i^* < u_{x_i} & \Rightarrow & \nu_i^* = 0 \end{array}\right\} \tag{3}$$

$$\left.\begin{array}{rcl} c_i^* = l_{c_i} & \Rightarrow & \lambda_i^* \geq 0 \\ c_i^* = u_{c_i} & \Rightarrow & \lambda_i^* \leq 0 \\ l_{c_i} < c_i^* < u_{c_i} & \Rightarrow & \lambda_i^* = 0 \end{array}\right\} \tag{4}$$

where $\mathbf{c}^* = \mathbf{c}(\mathbf{x}^*)$, $\mathbf{g}^* = \mathbf{g}(\mathbf{x}^*)$ etc.

The solver computes the maximum length

$$\mu_{\max} = \max_i \{ \|\mathbf{g}^*\|_2 \ , \ |\nu_i^*| \ , \ \|\mathbf{a}_i^*\|_2 |\lambda_i^*| \} \tag{5}$$

of the vectors that are summed in (1), and the normalized Kuhn-Tucker residual of (2) is defined by

$$r = \frac{\|\mathbf{g}^* - \boldsymbol{\nu}^* - \mathbf{A}^* \boldsymbol{\lambda}^*\|_2}{\max\{\mu_{\max}, 1.0\}}. \tag{6}$$

The solver terminates if the normalized Kuhn-Tucker residual satisfies $r \leq \epsilon$ where ϵ is a user provided tolerance. The use of the normalized Kuhn-Tucker residual removes scaling anomalies from the termination criterion.

The routine also terminates if the step $\|\mathbf{d}\|_\infty$ or the trust region radius ρ are less than ϵ. In these cases an assessment of the quality of the solution can be gained by also examining the maximum modulus Lagrange multiplier. If this is large, then it is possible that a solution has been obtained but the active constraints are nearly dependent.

2.2 Infeasible NLP problems

It may be that (P) has no point that satisfies the constraints or that the algorithm is unable to find a feasible point due to the nonconvex nature of (P). In this case, the algorithm is said to be in Phase I and aims to converge to a Kuhn–Tucker point of a feasibility problem

$$(F) \left\{ \begin{array}{ll} \displaystyle\min_x & \displaystyle\sum_{j \in J} c_j(\mathbf{x}) \\ \text{subject to} & c_j(\mathbf{x}) \leq 0 \ , \quad j \in J^\perp \end{array} \right.$$

where the index sets J and $J^\perp$ partition the nonlinear constraints into those that cannot be satisfied by the (QP) and those that can respectively. The solution of this feasibility problem indicates which constraints are causing the NLP (P) to be (locally) infeasible and the user can modify the problem accordingly. At a solution to (F) the values of the objective of (F), $h_J(\mathbf{x})$, and the sum of infeasibilities of the constraints of (F), $h_{J^\perp}(\mathbf{x})$, are passed back to the user. Conditions for termination in Phase I are analogous to those in Phase II (that is the SQP method).

The solver always checks the linear constraints first and terminates if they are inconsistent (`ifail = 2`).

2.3 Scaling

In many applications, variables of largely different magnitudes appear in the problem. The solver can be inefficient under these circumstances as the trust–region does not discriminate between variables of differing magnitudes. To compensate for this, the user may provide orders of magnitude of the variables or constraints or both and `filterSQP` will then solve a scaled problem.

Let s_i denote the order of magnitude of variable x_i, so that the scaled variable is $\hat{x}_i = x_i/s_i$. Letting $\mathbf{S} = \mathrm{diag}(s_1, \ldots, s_n)$ the "scaled" trust region becomes $\|\mathbf{S}^{-1}\mathbf{d}\| \leq \rho$ which means that in the scaled problem all variables are roughly of unit scale and a square trust region is appropriate.

Currently, scaling is done in such a way that the user provides all relevant information, such as function and derivative values in *unscaled* form. This information is then scaled automatically with the user provided scale factors before passing it into filterSQP.

The user needs to set `scale_mode` to indicate whether or not scaling is used. This is done in the common `scalec` (see Section 4.3). The scaling options available are: *(0)* no scaling, *(1)* user provided variable scaling, unit constraint scaling, *(2)* unit variable scaling, user provided constraint scaling, and *(3)* user provided variable scaling, user provided constraint scaling.

The constraint scaling also affects the computation of steepest edge coefficients in the QP solver `bqpd` and generally improves the performance of the QP solver at little additional cost.

3 System Requirements

The software package requires a `FORTRAN 77` compiler, the QP solver `bqpd` and a suitable dense or sparse linear algebra package (as provided with `bqpd`).

3.1 Mounting the Package on a `UNIX` workstation

The package comprises a suite of NLP subroutines:

`driver.f`	A sample driver for the NLP solver.
`filter.f`	The main SQP filter routine.
`filteraux.f`	Auxiliary routines used in `filter.f`.
`QPsolved.f`	The interface to the QP solver, dense storage.
`QPsolves.f`	The interface to the QP solver, sparse storage.
`scaling.f`	Routines that scale the problem.
`user.f`	The user supplied problem functions.

In addition the user requires a QP solver (`bqpd`) consisting of:

`bqpd.f`	The main QP solver routine.
`auxil.f`	Some auxiliary routines for `bqpd`.
`denseL.f`	Dense linear algebra package.
`sparseL.f`	Sparse linear algebra package.
`util.f`	Some linear algebra utilities.
`sparseA.f`	Sparse matrix storage/handling **OR**
`denseA.f`	Dense matrix storage/handling.

The user has the option of storing the matrix $\mathbf{A}^{(k)}$ either as a dense matrix (using `denseA.f`) or as a sparse matrix (using `sparseA.f`). Storage of the sparse matrices is explained in the routine `sparseA.f`. In addition, the user can choose a dense or sparse linear algebra solver (`denseL.f` or `sparseL.f`).

The solver takes advantage of the sparse data structure. For sparse problems of up to 100 variables there is no performance gain from a sparse solver, but exploiting the sparse data structure still makes sense. One attractive option is the possibility to use the sparse data structure with the dense solver. This is particularly useful for medium size problems which exhibit sparsity.

3.2 System Dependent Issues

3.2.1 Compiler Options

It is possible to compile the code in double precision using the flag `-r8` in the SUN f77 compiler. In this case, suitable values of tolerances for bqpd should be set. This can be done in the `block data defaults` following the routine bqpd, where examples of suitable values are also given. Changing from single to double precision is recommended for large or ill-conditioned problems.

An alternative to using the `-r8` flag is to make the following changes to all files:

1. Change all occurrences of `REAL` to `double precision`
2. Change all occurrences of `DOUBLE PRECISION` to `real*16`
3. Change all occurrences of `.E` to `.D`
4. Change all occurrences of `dble(·)` to a suitable routine that converts `double precision` to `real*16` type. On a SUN system, this routine is `qext(·)`.

Items 2 and 4 may be dispensed with, although the full value of iterative refinement may then not be realized. A small collection of shell-scripts for UNIX systems which perform these changes is available upon request from the authors.

3.2.2 System Dependent Routines

The file `driver.f` contains the `REAL function seconds` which calls a system dependent timing function to measure the CPU time taken by the solver. Suitable timing functions are provided for UNIX systems and (commented out) for Windows NT in `REAL function seconds`. The user must edit `REAL function` `seconds` for non-UNIX systems. Set `seconds = 1.E0` in `REAL function seconds` if no timing function is available (in this case CPU times will be zero for the solves).

The routine `function xlen` in `util.f` saveguards against overflow by using IEEE exception routines. When these are not available, an alternative form of `function xlen` can be used which is is not system dependent. This routine is provided in `util.f` but commented out.

4 Interface and User Supplied Routines

The interface of the NLP solver has the following form

```
      subroutine filterSQP (n,m,kmax,maxa,maxf,mlp,mxwk,mxiwk,iprint,
     .                     nout,ifail,rho,x,c,f,fmin,blo,bup,s,a,la,ws,
     .                     lws,lam,cstype,user,iuser,max_iter,istat,
     .                    rstat)
```

4.1 Definition of Parameters

A detailed description of the parameters follows below (the parameters preceded by a * must be set on entry to `filterSQP`.

*	`n`	number of variables (`INTEGER`)
*	`m`	number of constraints (excluding simple bounds) (`INTEGER`)
*	`kmax`	maximum size of null-space ($\leq$ `n`) (`INTEGER`)
*	`maxa`	maximum number of nonzero entries allowed in Jacobian matrix `a`, *only applies if* `sparseA.f` *is in use.* (`INTEGER`)
*	`maxf`	maximum size of the filter – typically 100 (`INTEGER`)
*	`mlp`	maximum level parameter for resolving degeneracy in `bqpd`; typically `mlp` $= 100$ (`INTEGER`)

*	`mxwk`	max. size of `REAL` workspace `ws` for SQP, QP and linear algebra solvers (`INTEGER`)
		For the *dense* linear algebra solver the amount of storage required is:
		`mxwk` $\leq$`16*n + 8*m + mlp + 8*maxf + kmax*(kmax+9)/2 + mxm1*(mxm1+3)/2`
		where `mxm1 = min(n,m+1)` is the maximum space allowed for `m1` in `denseL.f`
		For the *sparse* linear algebra solver the amount of storage required is:
		`mxwk` $\simeq$`16*n + 8*m + mlp + 8*maxf + kmax*(kmax+9)/2 + 5*n + nprof`
		where `nprof` is the space required for storing the row spikes of the L matrix and is *not* known a priori. A good guess for `nprof` is a multiple of `n`, say `20*n`.

*	`mxiwk`	max. size of `INTEGER` workspace `lws` for SQP, QP and linear algebra solvers (`INTEGER`)
		For the *dense* linear algebra solver the amount of storage required is:
		`mxiwk = 4*n + 3*m + mlp + 100 + kmax + mxm1`
		For the *sparse* linear algebra solver the amount of storage required is:
		`mxiwk = 4*n + 3*m + mlp + 100 + kmax +9*n+m`

*	`iprint`	print flag (`INTEGER`) for different amounts of print-out: 0 = quiet (no printing) 1 = one line per iteration 2 = scalar information printed 3 = scalar & vector information printed >3 = as 3, and call QP solver with `QPiprint = iprint-3`
*	`nout`	the output channel (6 = screen).

	`ifail`	fail flag (`INTEGER`) indicating successful run: -1 = ON ENTRY: warm start (use ONLY if `istat(1), n, m, lws` unchanged from previous call) 0 = successful run, solution found 1 = unbounded, feasible point $\mathbf{x}$ with $f(\mathbf{x}) \leq$ `fmin` found 2 = linear constraints are inconsistent 3 = (locally) nonlinear infeasible, optimal solution to feasibility problem (F) found 4 = terminate at point with $h(\mathbf{x}) \leq$ `eps` but QP infeasible 5 = termination with rho < `eps` 6 = termination with iter > max_iter 7 = crash in user routine (IEEE error) could not be resolved 8 = unexpect ifail from QP solver 9 = not enough REAL workspace 10 = not enough INTEGER workspace
*	`rho`	initial trust-region radius, default is 10 (`REAL`)
*	`x`	`x(n)` starting point and final solution (`REAL`)
	`c`	`c(m)` vector that stores the final values of the general constraints (`REAL`)
	`f`	the final objective value (`REAL`)
*	`fmin`	a lower bound on the objective value. The routine will terminate if a feasible $\mathbf{x}$ is found for which $f(\mathbf{x}) <$ `fmin` (`REAL`)
*	`blo`	`blo(n+m)` vector of lower bounds $(\mathbf{l}_x^T, \mathbf{l}_c^T)^T$ – simple bounds stored first (`REAL`)
*	`bup`	`bup(n+m)` vector of upper bounds similar to `blo` (`REAL`)
*	`s`	`s(n+m)` scale factors (see Section 2.3).
*	`a`	`a(1:a_entries)` stores the objective gradient and the constraint normals $[\mathbf{g} : \mathbf{A}]$ (`REAL`) *If using the sparse storage data structures* `n` *nonzero locations must be reserved for* $\mathbf{g}$ which is assumed to be stored in natural order. Further details on how to store sparse matrices are explained in Section 4.5.
	`la`	`la(0:a_entries+m+2)` column indices and length of rows of entries in `a` (`INTEGER`) (see Section 4.5). In the dense case `la` is treated as a scalar and gives the leading dimension of `a`.
	`ws`	`ws(mxwk)` `REAL` workspace
	`lws`	`lws(mxiwk)` `INTEGER` workspace
*	`lam`	`lam(n+m)` Lagrange multipliers of simple bounds and general constraints at solution (`REAL`)
*	`cstype`	`cstype(m)` indicates whether the constraint is linear or nonlinear, ie `cstype(j) = 'L'` for linear and `cstype(j) = 'N'` for nonlinear constraint number `j` (`CHARACTER*1`)
	`user`	real workspace, passed through to user routines such as gradient, hessian etc. (`REAL`)
	`iuser`	integer workspace, passed through to user routines such as gradient, hessian etc. (`INTEGER`)
*	`max_iter`	User supplied iteration limit for SQP solver (`INTEGER`)

`istat`	`istat(14)` `INTEGER` space for solution statistics:
	`istat(1)` = dimension of null space at solution
	`istat(2)` = number of iterations
	`istat(3)` = number of feasibility iterations
	`istat(4)` = number of objective evaluations
	`istat(5)` = number of constraint evaluations
	`istat(6)` = number of gradient evaluations
	`istat(7)` = number of Hessian evaluations
	`istat(8)` = number of QPs with mode ≤ 2
	`istat(9)` = number of QPs with mode ≥ 4
	`istat(10)` = total number of QP pivots
	`istat(11)` = number of SOC steps
	`istat(12)` = maximum size of filter
	`istat(13)` = maximum size of phase I filter
	`istat(14)` = number of QP crashes
`rstat`	`rstat(7)` `REAL` space for solution statistics:
	`rstat(1)` = l_2 norm of KT residual
	`rstat(2)` = $\mu_{\max}$ of (5)
	`rstat(3)` = largest modulus multiplier
	`rstat(4)` = l_∞ norm of final step
	`rstat(5)` = final constraint violation $h(\mathbf{x})$
	`rstat(6)` = $h_J(\mathbf{x})$, if `ifail = 1`
	`rstat(7)` = $h_{J^\perp}(\mathbf{x})$, if `ifail = 1`, see Section 2.2

4.2 Warm Starts

In certain situations, such as when solving MINLP problems by branch-and-bound, it is desirable to make use of information from a previous NLP solve. In this case, a warm start facility is available by setting `ifail = -1` on entry to `filterSQP`. This assumes that `istat(1), n, m, lws` are unchanged from this previous call.

4.3 Common Statements

A number of named common statement are used to pass information into `bqpd` and to pass global constants. These common statements take the following form

```
REAL                 infty, eps
common /NLP_eps_inf/ infty, eps
```

The common `/NLP_eps_inf/` defines ...

`infty`	A large number ∞ (default value is `1E20`)
`eps`	A tolerance ϵ (default value is `1E-6`), not the unit round-off

The tolerance ϵ is used in the termination criteria (Kuhn–Tucker error, nonlinear constraint residual and norm of the step).

```
REAL          ubd, tt
common /ubdc/ ubd, tt
```

`ubd` and `tt` define the upper bound on constraint violation used in the filter. The actual upper bound is defined by the maximum of `ubd` and `tt` times the initial constraint violation. Default values for `ubd, tt` are 100 and 0.125. On some problems, the solver re-enters feasibility restoration many times. In that case it may be better to use a tighter upper bound of say, `ubd = 10, tt = 0.0001`.

```
integer         char_l
character*10            pname
common /cpname/ char_l, pname
```

The common `cpname` defines the name of the problem and its length ...

`char_l`	The length of `pname` (default is `10`)
`pname`	The name of the NLP problem (default is `NLPproblem`)

The problem name is used to name the output files created by `filterSQP`. These files are only created if `iprint` ≥ 1 in `filterSQP`. The files are named ...

`*.summary`	A summary file with one line per iteration (see below).
`*.solution`	The solution of the problems $(\mathbf{x}^*, \boldsymbol{\lambda}^*)$ and the values of the constraints.
`*.output`	The output created by `SQPsolver`, unless `nout = 6` in which case output is written to the screen.
`*.outXXXX`	For problems with `n+m` ≥ 50 *and* `iprint` ≥ 3, `*.output` contains output only for the first iteration. Subsequent iterations are written to `*.outXXXX`, where `XXXX` is the iteration number. This is to prevent the files getting too large for larger problems.

```
integer         scale_mode, phe
common /scalec/ scale_mode, phe
```

The common `scalec` passes the parameter `scale_mode` There are four modes of scaling currently supported:

mode		type of scaling
0	=	no scaling
1	=	user provided variable scaling
2	=	unit variable scaling, user provided constraint scaling
3	=	user provided variable and constraint scaling

4.4 Output

If `iprint` is greater than 0, then `filterSQP` creates three output files: `*.summary`, `*.output` and `*.solution`, where `*` stands for the name of the problem supplied in `pname`. If `nout = 6` then `*.output` is written to the screen. Otherwise, it is written to a file. Note that `nout` should not be set to 1 or 2, as these are used for `*.solution` and `*.summary`.

The first file is a summary of the progress of the SQP solver with one line per iteration. It gives iteration number `iter`, the trust–region radius `rho`, the step length `||d||`, the values of the objective `f / hJ`, the constraint violation `||c||/hJt`, the penalty function `penalty fcn` and finally a 5 character long string that indicates the type of iteration `IS`.

The string `IS` has the format ±`POXX` where + indicates a successful step and − an unsuccessful step, `P` is the phase (either 1 for feasibility iteration or 2 for normal SQP), `O` gives the order of the step (1 = LP, 2 = QP, 3 = Second order correction and 4 = unblocked). Finally `XX` is nonblank if any heuristics were activated during the step (`UB` = upper bound, `NW` = North West corner rule, `SE` = South East corner rule). See [3] for a detailed description of the various heuristics.

If a QP problem causes `bqpd` to crash (with `bqpd.ifail = 8`, then `XX` is set to `-8`. The number of these crashes is recorded in `istat(14)`. Following a crash of `bqpd` the trust-region radius is reduced and the QP problem is re-solved in cold-start mode.

The file `*.output` is self explanatory and contains further details of the run (if `iprint` ≥ 2). The file `*.solution` contains the final solution $\mathbf{x}^*$ and $\mathbf{c}^*$ with lower bound, value, upper bound, Lagrange multiplier.

4.5 User defined Subroutines

All function evaluations, gradient and Hessian computations are the sole responsibility of the user. The following routines are used in filter to evaluate these functions. These user supplied function can be found in the file `user.f`. Subroutines that interface filterSQP to CUTE and AMPL are also available upon request.

```
subroutine confun(x, n, m, c, a, la, user, iuser, flag)
```

`confun` evaluates the constraint values (both linear and nonlinear). The parameters are ...

`x`	`x(n)` the value of the current variables (input from `filterSQP`) (`REAL`)
`n`	number of variables (`INTEGER`)
`m`	number of linear & nonlinear constraints (excluding simple bounds) (`INTEGER`)
`c`	`c(m)` vector that stores the values of the constraints (output) (`REAL`)
`a`	the Jacobian matrix (passed through from `filterSQP` to evaluate the linear constraints) (`REAL`)
`la`	indexing information relating to `a` (`INTEGER`)
`user`	user workspace (see above)
`iuser`	user workspace (see above)
`flag`	Set to 1 if arithmetics exception occurred in `confun`, 0 otherwise.

```
      subroutine objfun(x, n, f, user, iuser, flag)
```

`objfun` evaluates the objective function value. The parameters are ...

`x`	`x(n)` the value of the current variables (input from filter) (`REAL`)
`n`	number of variables (`INTEGER`)
`f`	the values of the objective function (output) (`REAL`)
`user`	user workspace (see above)
`iuser`	user workspace (see above)
`flag`	Set to 1 if arithmetics exception occurred in `objfun`, 0 otherwise.

```
subroutine gradient(n,m,mxa,x,a,la,maxa,user,iuser,flag)
```

`gradient` evaluates the objective gradient and the constraint Jacobian matrix. It is important to always leave `n` spaces for the objective gradient, as this is used in the feasibility restoration step.

Note: Linear constraint gradients need not be re–computed. However, the user is responsible for keeping this information consistent. One way of achieving this is to use a routine `initialize_NLP` to initialize the linear constraints.

`n`	number of variables (`INTEGER`)
`m`	number of constraints (`INTEGER`)
`mxa`	actual number of entries in `a` (`INTEGER`) (returned for information only)
`x`	`x(n)` the value of the current variables (input from filter) (`REAL`)
`a`	the Jacobian vector storing the nonzeros of the Jacobian (`REAL`)
`la`	`la(0:*)` column indices for `a` and pointers to start of each row (`INTEGER`)
`maxa`	maximum size of `a` (`INTEGER`)
`user`	user workspace (see above)
`iuser`	user workspace (see above)
`flag`	Set to 1 if arithmetics exception occurred in `objfun`, 0 otherwise. If an arithmetic exception occurred, then the gradients must *not* be modified by `gradient`.

The sparse storage scheme is explained in the sparse linear algebra subroutines, which is reproduced here for the sake of completeness.

The sparse matrix data structure stores as a set of column vectors the following matrix:

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

where $\mathbf{g}$ is the current gradient of f and $\mathbf{A}$ is the Jacobian of the constraints $\mathbf{c}$. The number of nonzeros in this matrix is `nnza`.

The matrix $\hat{\mathbf{A}}$ contains gradients of the linear terms in the objective function (column 0) and the general constraints (columns 1:m). No explicit reference to simple bound constraints is required in $\hat{\mathbf{A}}$. The information is set in the parameters `a` (`REAL`) and `la` (`INTEGER`) of `filterSQP`.

In this sparse format, these vectors have dimension `a(1:nnza)` and `la(0:lamax)`, where `nnza` is the number of nonzero elements in $\hat{\mathbf{A}}$, and `lamax` is at least `nnza+m+2`. The last `m+2` elements in `la` are pointers.

The vectors `a(.)` and `la(.)` must be set as follows:

`a(j)` and `la(j)` for `j=1,nnza` are set to the values and row indices (respectively) of all the nonzero elements of $\hat{\mathbf{A}}$. Entries for each column are grouped together in increasing column order.

`la(0)` points to the start of the pointer information in `la`. `la(0)` must be set to `nnza+1` (or a larger value if it is desired to allow for future increases to `nnza`).

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

```
      subroutine hessian (x,n,m,phase,lam,ws,lws,user,iuser,l_hess,
     .                    li_hess,flag)
```

The subroutine `hessian` resets the Hessian information that is stored in `ws` and `lws` and passed through `bqpd` to `Wdotd`. The user must reset the Hessian of the Lagrangian $\mathbf{W}^{(k)}$ for the values of $\mathbf{x}$ (`x`) and $\boldsymbol{\lambda}$ (`lam`). The Hessian may be stored in any form convenient for the problem in hand; but note that `Wdotd` must be kept consistent.

`x`	`x(n)` the value of the current variables (input from filter) (`REAL`)
`n`	number of variables (`INTEGER`)
`m`	number of constraints (`INTEGER`)
`phase`	indicates what kind of Hessian matrix is required. `phase = 2` Hessian of the Lagrangian, `phase = 1` Hessian of the Lagrangian *without* the objective Hessian. (`INTEGER`)
`lam`	`lam(n+m)` vector of Lagrange multipliers (`REAL`)
`ws`	workspace for Hessian, passed to `Wdotd` (`REAL`)
`lws`	workspace for Hessian, passed to `Wdotd` (`INTEGER`)
`user`	user workspace (see above)
`iuser`	user workspace (see above)
`l_hess`	*On entry:* max. space allowed for Hessian storage in `ws`. *On exit:* actual amount of Hessian storage used in `ws` (`INTEGER`).
`li_hess`	*On entry:* max. space allowed for Hessian storage in `lws`. *On exit:* actual amount of Hessian storage used in `lws` (`INTEGER`).
`flag`	Set to 1 if arithmetics exception occurred in `hessian`, 0 otherwise. If an arithmetic exception occurred, then the Hessian must *not* be modified by `hessian`.

In our routine `hessian` we use a common that constitutes a storage map for `lws` which is used to pass Hessian information through the QP solver to `Wdotd` (which is explained below).

```
c     ... storage map for Hessian storage
      integer        phl, phr, phc
      common /hessc/ phl, phr, phc
```

The three pointers for `lws` are:

`phl`	pointer to location in `lws`, which stores the number of Hessian entries (sparse Hessian).
`phr`	pointer to start of row indices of Hessian in `lws`.
`phc`	pointer to start of column indices of Hessian in `lws`.

Thus, in our Hessian implementation, `ws(i)` is the Hessian value for row `lws(phr+i)` and column `lws(phc+i)` for `i=1:lws(phl)`.

Note: The user can choose to store the Hessian information in any form (e.g. for certain application an outer product form may be more efficient), as long as `hessian` and `Wdotd` are consistent.

The user is also responsible for providing a subroutine `Wdotd` which computes the product $\mathbf{v} = \mathbf{W}^{(k)} \cdot \mathbf{d}$ of the Hessian matrix $\mathbf{W}^{(k)}$ with an arbitrary vector $\mathbf{d}$. The header of the routine is ...

```
      subroutine Wdotd(n, d, ws, lws, v)
c     ... user subroutine to compute v = W.d
      integer n
      REAL    d(n), ws(*), v(n)
      integer lws(*)
```

Information from the Hessian evaluation `hessian` is passed through the QP solver to this routine in the workspaces `ws, lws`. The user can choose his/her own Hessian representation and storage (see `Wdotd` for an example).

5 NLP Example Problem

In the distribution an example problem is included which is taken from A. Duran and I.E. Grossmann [2]. The problem is defined in the user routines `user.f` and the input file `test1.s`. A detailed problem description is given below.
Test problem TP1

$$\left\{\begin{array}{lll}
\min\limits_{x,y} & 5y_1 + 6y_2 + 8y_3 + 10x_1 - 7x_3 - 18\ln(x_2+1) & \\
 & -19.2\ln(x_1 - x_2 + 1) + 10 & \\
\text{subject to} & 0.8\ln(x_2+1) + 0.96\ln(x_1-x_2+1) - 0.8x_3 & \geq 0 \\
 & \ln(x_2+1) + 1.2\ln(x_1-x_2+1) - x_3 - 2y_3 & \geq -2 \\
 & x_2 - x_1 & \leq 0 \\
 & x_2 - 2y_1 & \leq 0 \\
 & x_1 - x_2 - 2y_2 & \leq 0 \\
 & y_1 + y_2 & \leq 1 \\
 & 0 \leq x \leq u, \quad \text{where } u^T = (2,2,1) & \\
 & 0 \leq y_i \leq 1 & \\
\text{NLP solution} & f^* = 0.759, \ y^* = (0.273, 0.300, 0.000)^T & \\
 & x^* = (1.147, 0.547, 1.000)^T &
\end{array}\right.$$

6 Interfaces to AMPL, CUTE, C/C++

We can provide interfaces to AMPL or CUTE [1] upon request. AMPL is an algebraic modelling language. CUTE allows the user to specify a problem in the SIF format and can be obtained from `http://www.dci.clrc.ac.uk/Activity.asp?CUTE`.

All codes are written in Fortran 77. However, we have had no trouble interfacing them to C code using the Fortran to C converter `f2c` which is available at `http://netlib.bell-labs.com/netlib/f2c/`.

Note that while the SIF format allows the user to provide scale factors, there are no tools in CUTE to extract these scale factors from the `OUTSDIF.d` file, generated by CUTE. As a consequence, scaling has to be done within the CUTE file.

7 Changes to earlier Versions

Date	Changes
June 1998	Parameter `istat` now has 13 entries.
	Record crashes of bqpd `ifail = 9`.
	Facility for changing to double precision added.
September 1998	Modified termination criterion (scaled).
November 1998	Added common `ubdc` for upper bound.
January 1999	Added warm start for NLP solver.
March 1999	(1) `ifail` indicator changed.
	(2) `istat` and `rstat` changed statistics, print using `print_stats`.
	(3) new subroutine `readpar` to read user parameters for solve.
	(4) Added AMPL/CUTE/C/C++ interface section.

References

[1] Bongartz, I. Conn, A.R. Gould, N.I.M. and Toint, Ph.L. CUTE: Constrained and Unconstrained Testing Enviroment. *ACM Transactions on Mathematical Software*, (21):123–160, 1995.

[2] Duran, M. and Grossmann, I.E. An outer-approximation algorithm for a class of mixed–integer nonlinear programs. *Mathematical Programming*, 36:307–339, 1986.

[3] Fletcher, R. and Leyffer, S. Nonlinear programming without a penalty function. Numerical Analysis Report NA/171, Department of Mathematics, University of Dundee, September 1997.