The NLP solver `filterSD`

1. Introduction

The file `filterSD.f` contains a Fortran 77 subroutine that aims to find a solution of the NLP (Nonlinear Programming Problem)

$$\begin{array}{ll} \text{minimize} & f(\mathbf{x}) \\ \text{subject to} & \mathbf{l} \leq \begin{bmatrix} \mathbf{x} \\ \mathbf{c}(\mathbf{x}) \end{bmatrix} \leq \mathbf{u} \end{array}$$

in double length arithmetic. Here $f(\mathbf{x})$ is a given objective function of n variables $\mathbf{x}$, and $\mathbf{c}(\mathbf{x})$ is a vector of m constraint functions. It is required that these functions are continuously differentiable at points that satisfy the bounds on $\mathbf{x}$, and that the user is able to compute the gradient vector $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})$ and the Jacobian matrix $A(\mathbf{x}) = \nabla \mathbf{c}^T$. Lower and upper bound constraints on the variables $\mathbf{x}$ and the constraint functions $\mathbf{c}(\mathbf{x})$ may be supplied.

There main design aims of the code have been to avoid the use of second derivatives, and to avoid storing an approximate reduced Hessian matrix by using a new limited memory spectral gradient approach based on Ritz values.

The basic approach is that of Robinson's method, globalised by using a filter and trust region. The code calls the Linear Constraint Problem (LCP) solver `glcpd.f` which has been developed using the Ritz values approach. A generalisation of this idea is used to obtain feasibility in the NLP problem. However it is possible that the code might terminate at a locally infeasible point, which is a local minimizer of the L1 sum of general constraint infeasibilities $h(\mathbf{x})$, subject to the bounds on the variables.

2. Calling Sequence and Parameter List

The calling sequence for the subroutine is

`call filterSD(n, m, x, al, f, fmin, cstype, bl, bu, ws, lws, v, nv, maxa, maxla,`

`maxu, maxiu, kmax, maxg, rho, htol, rgtol, maxit, iprint, nout, ifail)`

Note that Fortran 77 implicit declarations are used, and real variables are implemented in `double precision`.

The description of the parameters is as follows

`n` number of variables
`m` number of general constraints

`x(n+m)` The vector of variables is stored in `x(1:n)`. Initially an estimate of the solution must be set, replaced by the solution (if it exists) on exit. The rest of `x` is workspace

`al(n+m)` stores Lagrange multipliers of simple bounds and general constraints at the solution on exit. A positive multiplier indicates that the lower bound is active, and a negative multiplier indicates that the upper bound is active. Inactive constraints have a zero multiplier.

`f` returns the value of $f(\mathbf{x})$ when $\mathbf{x}$ is a feasible solution

`fmin` set a strict lower bound on $f(\mathbf{x})$ for feasible $\mathbf{x}$ (used to identify an unbounded NLP)

`cstype(m)` character workspace: if `ifail` = 3, `cstype` indicates constraints that are infeasible in the L1 solution. `cstype(i)` = `A` if the lower bound on constraint `i` is infeasible, `Z` if the upper bound is infeasible, else `N` if feasible.

`bl(n+m)` lower bounds on $\mathbf{x}$ and $\mathbf{c}(\mathbf{x})$ (use numbers no less than `-ainfty` (see below), and where possible supply realistic bounds on $\mathbf{x}$)

`bu(n+m)` upper bounds on $\mathbf{x}$ and $\mathbf{c}(\mathbf{x})$ (use numbers no greater than `ainfty`)

`ws(*)` double precision workspace

`lws(*)` integer workspace

`v(maxg)` stores `nv` Ritz values (estimates of eigenvalues of the reduced Hessian): supply the setting from a previous run of `filterSD`, or set `nv` = 1 and `v(1)` = `1.D0` in absence of other information

`nv` number of values set in `v`

`maxa` maximum number of entries in the Jacobian `a(*)` set by 'gradients'

`maxla` number of locations required for sparse matrix indices and pointers `la(0:*)` to be set up in `lws(*)` (`maxla` $\geq$ `maxa` $+ m + 3$). Set `maxla` = 1 if using dense matrix format

`maxu` length of workspace `user(*)` passed through to user subroutines 'functions' and 'gradients'

`maxiu` length of workspace `iuser(*)` passed through to user subroutines

`kmax` maximum dimension of null space allowed for (`kmax` $\leq n$)

`maxg` maximum number of reduced gradient vectors stored by the limited memory method (typically 6 or 7)

`rho` initial trust region radius (typically 1.D1)

`htol` tolerance allowed in the sum $h(\mathbf{x})$ of constraint infeasibilities (e.g. 1.D-6)

`rgtol` tolerance allowed in reduced gradient L2 norm (typically 1.D-4)

`maxit` maximum number of major iterations allowed

`iprint` verbosity of printing (0=none, 1=one line per iteration, 2=additional text information given)

`nout` output channel for printing

`ifail` returns failure indication as follows

0 = successful run
1 = unbounded NLP ($f(\mathbf{x}) \leq$ `fmin` at an `htol`-feasible point $\mathbf{x}$)
2 = bounds on $\mathbf{x}$ are inconsistent
3 = local minimum of feasibility problem and $h(\mathbf{x}) >$ `htol` (nonlinear constraints are locally inconsistent)
4 = initial point x has $h(\mathbf{x}) >$ `ubd` (reset `ubd` or $\mathbf{x}$ and re-enter)
5 = `maxit` major iterations have been carried out
6 = termination with `rho` $\leq$ `htol`
7 = not enough workspace in `ws` or `lws` (see message)
8 = insufficient space for filter (increase `mxf` and re-enter)
> 9 = unexpected fail in LCP solver (10 has been added to the LCP `ifail`)

3. User Subroutines

The user must provide two subroutines to calculate $f(\mathbf{x})$, $\mathbf{c}(\mathbf{x})$ and their first derivatives as follows

```
subroutine functions(n, m, x, f, c, user, iuser)
implicit double precision (a − h, o − z)
dimension x(∗), c(∗), user(∗), iuser(∗)
...
```

Statements to compute $f(\mathbf{x})$ and the $m-$vector $\mathbf{c}(\mathbf{x})$ in `f` and `c` from $\mathbf{x}$. The arrays `user` and `iuser` enable the user to pass information from the driver to the subroutine. The user is responsible for ensuring that any failures such as IEEE errors (overflow, NaN′s etc.) are trapped and not returned to `filterSD` The same holds for gradients.

```
...
return
end

subroutine gradients(n, m, x, a, user, iuser)
implicit double precision (a − h, o − z)
dimension x(∗), a(∗), user(∗), iuser(∗)
...
```

Statements to calculate gradients of $f(\mathbf{x})$ and $\mathbf{c}(\mathbf{x})$. The column vector $\nabla f(\mathbf{x})$ must be followed by the column vectors $\nabla c_i(\mathbf{x})$, $i = 1, 2, \ldots, m$ in the one dimensional array `a`$(*)$. Either a dense or sparse data structure may be used. If using the sparse data structure, only structurally nonzero entries are set. Pointers etc. for the data structure are set elsewhere in `lws`. Any call of `gradients` immediately follows one of `functions` with the same $\mathbf{x}$.

```
...
return
end
```

The user must also supply a driver routine which calls `filterSD`. This must set parameters and common blocks of `filterSD` as appropriate. Space for `x,al,bl,bu,ws, lws,v` and `cstype` must be assigned. If using the sparse data structure for setting gradients, indices and pointers `la(0:maxla-1)` must be set by the driver in `lws`, immediately following any user workspace in `lws(1:maxiu)`. No changes in this data structure are allowed during the operation of `filterSD`. For dense format just set `maxla` = 1 and set `lws(maxiu+1)` to the 'stride' ($\geq n$) used in setting the columns of ∇f and ∇c_i. For efficiency, constant entries in the gradient parameter `a(*)` above may be set once and for all in the driver. These must be set in the workspace array `ws(*)`. However two copies of `a(*)` are kept by `filterSD`. These reside in `ws(maxu+1:maxu+maxa)` and `ws(maxu+maxa+1:maxu+2*maxa)`. Any constant entries must be set in both copies.

Precise details of the format of `a(*)` and `la(*)` are described in Section 6 of the document `glcpd.pdf`.

4. Common blocks

`common/wsc/kk,ll,kkk,lll,mxws,mxlws`

The user must specify the length of the workspace arrays `ws(*)` and `lws(*)` in `mxws` and `mxlws` respectively. It may not be easy to specify a-priori how large these arrays should be. Set a suitable estimate, and `filterSD` will prompt if larger values are required. As a guide, `ws(*)` contains first user workspace, then workspace for `filterSD`, then workspace for `glcpd`, and finally workspace for `denseL.f` or `schurQR.f`. `lws(*)` contains user workspace, then `maxla+n+m+mlp` locations for `filterSD` and additional locations for `denseL.f` or `schurQR.f`.

`common/defaultc/ainfty,ubd,mlp,mxf`

Default values of some control parameters are set here. `ainfty` is used to represent infinity. `ubd` provides an upper bound on the allowed constraint violation. `mlp` is the maximum length of arrays used in degeneracy control. `mxf` is the maximum length of filter arrays. Default values are 1.D20, 1.D4, 50, 50.

`common/ngrc/mxgr`

The user can limit the time spent in each call of the LCP solver by setting an upper limit (e.g. 100) on the number of gradient calls in `mxgr` (default=1000000).

`common/mxm1c/mxm1`

When using `denseL.f`, `mxm1` must be set to the maximum number of general constraints allowed in the active set. `mxm1` = $\min(m+1, n)$ is always sufficient. `mxm1c` can be omitted when using `sparseL.f`.

`common/epsc/eps,tol,emin`

`common/repc/sgnf,nrep,npiv,nres`

`common/refactorc/mc,mxmc`

These common blocks provide default parameters that control `glcpd`. The default value of `mxmc=500` is not suitable for use when using `schurQR.f` or `sparseL.f` and should be reset to a smaller value (typically 25).

`common/statsc/dnorm,h,hJt,hJ,ipeq,k,itn,nft,ngt`

This common block returns information about the outcome of `filterSD`. `dnorm`=final step length, `h`=final constraint violation, `hJt`=ditto for 'N' constraints, `hJ`=ditto for 'A' and 'Z' constraints, `ipeq`=number of active equations, `k`=number of free variables, `itn`=number of iterations, `nft`=total number of function calls, `ngt`=total number of gradient calls.

5. Checking Derivatives

It is very easy to make errors when deriving formulae for the first derivatives $\nabla f(\mathbf{x})$ and $\nabla c_i(\mathbf{x})$, $i = 1, 2, \ldots, m$. Such errors will almost certainly cause `filterSD` to malfunction, probably in an unpredictable way. Therefore the user is strongly advised to use the derivative checking subroutine `checkd` which is located in the file `checkd.f`. This is done by including a call to `checkd` in the driver, immediately before the call to `filterSD`. The parameters of `checkd` are a subset of those for `filterSD` so this is easily done. The only extra information to provide is some non-zero finite difference intervals h_i, $i = 1, 2, \ldots, n$. This is done in the parameter `al(1:n)`. Typical values would be say 10^{-2} or 10^{-3} times the typical magnitudes of the variables x_i. The program checks that the difference quotient in each coordinate direction lies between the derivatives at the ends of the interval. If an inconsistency is found then the code reports the constraint and variable and exits (constraint 0 refers to the objective function). A successful check is recognised by the messages 'entering checkd' followed by 'exiting checkd', whereupon the call to `checkd` can be commented out. The code is suitable for use with both sparse or dense matrix formats without change. Further details are given in the file `checkd.f`. Illustrations are provided in the example programs.

6. An Illustrative Example

Driver programs are provided to solve the Hock-Schittkowski NLP problem HS106, using either sparse or dense matrix formats. They illustrate various aspects such as provision of subroutines, setting indices, pointers and constant values relating to the gradient and Jacobian matrix, passing information through to the subroutines, and checking derivatives. The drivers are located in the files `hs106.f` and `hs106d.f`.

7. Output

If `iprint` = 0, no output from the code is made. If `iprint` = 1, one line of output is given for each iteration or failed iteration. Presence of the symbols '`<<`' indicates

that iterations are in phase 2, and columns 2 and 3 of the output refer to the values of $h(\mathbf{x})$ and $f(\mathbf{x})$ respectively. Otherwise the symbol '`<`' appears, iterations are in feasibility restoration mode (phase 1), and columns 2 and 3 refer to the values `hJt` and `hJ` respectively (see the end of Section 4 above). If the text '`feasible LP`' appears then the previous iteration is not acceptable (e.g. to the filter) and `rho` is reduced and an LP subproblem is solved, giving a new multiplier estimates. If the text '`project`' appears then the previous step is not acceptable, but the method is taking a projection step with the aim of finding an acceptable iterate with smaller constraint violation. If `iprint` = 2, then further messages are included in the output describing the progress of the iterations.

8. Performance and Troubleshooting

Extensive testing of the code indicates that a high percentage of test problems (mostly from CUTEr) have been solved effectively. However some negative features have been observed and were indeed expected. When the dimension of the null space is not small, then the lack of reduced Hessian information necessitates many more gradient evaluations to solve each LCP problem. Likewise, although Ritz values are passed from one LCP call to the next, these contain much less information than would be provided by the reduced Hessian matrix. Thus the number of function and gradient calls can be much larger than is required by some other codes that provide approximate or exact reduced Hessian information. To some extent this can be alleviated by using the parameter `mxgr` to limit the number of gradient calls made by the LCP solver. Another possibility is to relax the tolerance `rgtol` on the reduced gradient. In practice it is often the case that the cost of a function or gradient evaluation is much less than other costs involved in solving the LCP and LP subproblems, and overall run times seem to be very reasonable, as does reliability.

Slow convergence can sometimes be caused by too small a value of `ubd`, but on the other hand, too large a value may allow iterates to visit highly infeasible iterates from which it is difficult to recover. Clearly this is a matter for experimentation. Choice of the initial trust region `rho` (whose aim is to restrict the size of step to regions in which the current algorithm model is valid) can be important, but the code is usually available to adjust `rho` to a satisfactory value. It is important to choose units in which all variables have a similar effect on the values of the objective and constraint functions.

The code is designed with the aim of converging rapidly in the neighbourhood of a point of local infeasibility. In fact a post-processor `l1sold` to `glcpd` (or `qlcpd`), located in the file `l1sold.f`, has been written to find the best L1 solution of infeasibilities when `glcpd` recognises that an LCP is infeasible, so that the solution returned by `filterSD` is a non-zero local minimizer of the L1 sum of infeasibilities.

The user might ask what can be done if `filterSD` returns with a certificate of local infeasibility (`ifail` = 3). This is not easy to resolve, since finding feasibility is a global optimization problem, and hence is usually intractable for other than small problems. One thing to suggest is to examine the output to see if a feasible or near feasible point has been visited on an earlier iteration. In this case, re-starting from such a point with a smaller value of `ubd` should be tried. Another suggestion is to check if the subset of all linear constraints is consistent. This can be done by relaxing the bounds on the nonlinear constraints to $(-\infty, \infty)$ and repeating. Otherwise the only advice I can offer is to re-run with different starting points or parameter values. One way of doing this (not guaranteed to work, of course) is to relax the bounds on all the nonlinear constraints by a smallish amount such that `filterSD` returns a feasible solution. Then re-run from this solution with the bounds restored.

Other causes of failure may also be difficult to resolve. The importance of checking gradient formulae has already been referred to. It can also be important to pay attention to the conditioning of the problem, for example by choosing a formulation in which the Jacobian matrix is not close to becoming singular or rank-deficient. In particular, it is advisable to pay attention to the scaling of constraints and variables. Thus the units in which the variables are expressed should be chosen to be of similar magnitude, and preferably of an order of magnitude of around unity. The same is true of the typical values of the constraint functions. It is under these circumstances that the heuristics used by the code should be most effective. Another piece of advice is to avoid using bounds of $\pm\infty$ on the variables, when tighter and more realistic bounds are available. This can prevent the code from visiting unproductive regions of design space, as the code always respects the simple bounds on the variables, even during feasibility restoration.

Other causes of failure may be caused by malfunction in the LCP solver. The document `glcpd.pdf` describes some possible reasons and how they might be resolved.