The LCP solvers `qlcpd` and `glcpd`

1. Introduction

The file `glcpd.f` contains a Fortran 77 subroutine that aims to find a solution of the General LCP (Linearly Constrained Problem)

$$\begin{aligned} &\text{minimize} \quad f(\mathbf{x}) \\ &\text{subject to} \quad \mathbf{l} \le \begin{bmatrix} \mathbf{x} \\ A^T\mathbf{x} \end{bmatrix} \le \mathbf{u} \end{aligned}$$

in double length arithmetic. Here $f(\mathbf{x})$ is a given differentiable function of n variables $\mathbf{x}$, and it is required that the user is able to compute the gradient vector $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})$. Lower and upper bound constraints on the variables $\mathbf{x}$ and m linear functions $A^T\mathbf{x}$ may be supplied, where A is an $n \times m$ matrix. A recursive form of an active set method is used, using Wolfe's method to resolve degeneracy. A limited memory reduced gradient sweep method is used for minimization in the null space. Matrix information is made available and processed by calls to external subroutines. Details of these are given in auxiliary files named either `denseA.f` and `denseL.f` or `sparseA.f` and `schurQR.f`. The latter replaces an earlier file `sparseL.f`. The utilities file `util.f` is also required.

The code can also be used to solve LP, QP, and unconstrained optimization problems (possibly with simple bounds on the variables), for which it should be reasonably effective. However the file `qlcpd.f` contains a version of `glcpd.f` specifically designed to solve QP and LP problems.

2. Calling Sequence and Parameter List

The calling sequence for the `glcpd` subroutine is

```
call glcpd(n, m, k, kmax, maxg, a, la, x, bl, bu, f, fmin, g, r, w, e, ls,
* alp, lp, mlp, peq, ws, lws, cws, v, nv, rgtol, m0de, ifail, mxgr, iprint, nout)
```

Note that Fortran 77 implicit declarations are used, and initial letters `p` and `q` declare `integer` variables rather than the more usual `double precision`. Details for `qlcpd` are similar and are described in the file `qlcpd.f`.

The description of the parameters is as follows

- `n` number of variables
- `m` number of general constraints (columns of A)
- `k` dimension k of the null space obtained by eliminating the active constraints (`k` is only to be set if `mode` ≥ 2). The number of constraints in the active set is $n - k$

`kmax` maximum value of k (`kmax` $\le n$)

`maxg` maximum number of reduced gradient vectors stored in sweep method: ($1 <$ `maxg` $\leq$ `kmax` $+ 1$ when `kmax` > 0), typically `maxg` $= \min(6,$ `kmax` $+ 1)$

`a(*)` storage of double precision reals associated with A. This storage may be provided in either dense or sparse format. Refer to Section 6 below for information on how to set `a(*)` and `la(*)`.

`la(*)` storage of integers associated with A

`x(n)` contains the vector of variables. Initially an estimate of the solution must be set, replaced by the solution (if it exists) on exit.

`bl(n+m)` vector of lower bounds for variables and general constraints

`bu(n+m)` vector of upper bounds (use numbers less than about `1.D20`, and where possible supply realistic bounds on the x variables)

`f` returns the value of $f(\mathbf{x})$ when $\mathbf{x}$ is a feasible solution. If `ifail` $= 3$ below, `f` stores the sum of constraint infeasibilities

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

`g(n)` returns the gradient vector of $f(\mathbf{x})$ when $\mathbf{x}$ is feasible

`r(n+m)` workspace: stores residual vectors and multipliers. For $j = n+1, \ldots, n+m$ the value of `r(abs(ls(j)))` stores the residual of an inactive variable or constraint `i` where `i=abs(ls(j))` (see `ls` below). The sign of `ls(j)` tells from which bound the residual is measured. For $j = 1, \ldots, n$ the value of `r(i)` tells the multiplier of the active constraint or free variable `i`. The sign convention is such that residuals are nonnegative at a solution (except for multipliers of free variables (these `r(i)` are the *reduced gradients*) or equality constraints)

`w(n+m)` workspace: stores denominators for ratio tests

`e(n+m)` stores steepest-edge normalization coefficients: if `mode` > 2 then information in this vector from a previous call should not be changed. (In mode 3 these values provide approximate coefficients)

`ls(n+m)` stores indices of the active constraints in locations $1 : n$ and of the inactive constraints in locations $n+1 : n+m$. The simple bounds on the variables are indexed by $1 : n$ and the general constraints by $n+1 : n+m$. The sign of `ls(j)` indicates whether the lower bound ($+$) or the upper bound ($-$) of constraint `ls(j)` is currently significant. Within the set of active constraints, locations $1 :$ `peq` store the store indices of active equality constraints and locations `peq` $+ 1 : n - k$ store indices of active inequality constraints. Finally locations $n - k + 1 : n$ store indices of any free variables (these are variables not on a bound, which are used to parametrise the null space: `ls(j)` is always positive in this range). If `mode` ≥ 2, the first $n - k$ elements of `ls` must be set on entry to `glcpd`.

`alp(mlp)` workspace associated with recursion

`lp(mlp)` list of pointers to recursion information in `ls`

`mlp` maximum number of levels of recursion allowed (`mlp` > 2: typically `mlp` $= 50$ would usually be adequate but `mlp` $= m$ is an upper bound)

`peq` pointer to the end of equality constraint indices in `ls`

`ws(*)` real workspace for `funct`, `grad`, `glcpd` and `denseL.f` (or `schurQR.f`). Set the total number (or a larger number) in `mxws` (see Section 4 below).

`lws(*)` integer workspace for `funct`, `grad`, `glcpd` and `denseL.f` (or `schurQR.f`). Set the total number (or a larger number) in `mxlws` (see Section 4 below). The storage maps for `ws` and `lws` are set by the subroutine `stmap`.

`cws(*)` character workspace (if any) needed by `funct`

`v(maxg)` set `nv` estimates of the eigenvalues of the reduced Hessian of $f(\mathbf{x})$ (for example from a previous run of `glcpd`). Set `nv` $= 1$ and `v(1)` $=$ `1.D0` in absence of other information. New values of `v` are left on exit.

`nv` Number of estimates in `v`

`rgtol` required accuracy in the reduced gradient $L2$ norm: it is advisable not to seek too high accuracy - `rgtol` may be increased by the code if it is deemed to be too small, see the definition of `sgnf` in Section 5.

`m0de` mode of operation (larger numbers imply extra information):

- 0 = cold start (no other information available, takes simple bounds for the initial active set)
- 1 = as 0 but includes all equality constraints in initial active set
- 2 = user sets $n - k$ active constraint indices in `ls(j)`, $j = 1, \ldots, n - k$. For a general constraint the sign of `ls(j)` indicates which bound to use. For a simple bound the current value of $\mathbf{x}$ is used
- 3 = takes active set and other information from a previous call. Steepest edge weights are approximated using previous values.
- 4 = as 3 but it is also assumed that columns of A are unchanged so that factors of the basis matrix stored in `ws` and `lws` are valid (changes in $f(\mathbf{x})$ and the vectors $\mathbf{l}$ and $\mathbf{u}$ are allowed)

A local copy (`mode`) of `m0de` is made and may be changed by `glcpd`

`ifail` outcome of the process

- 0 = cold start (no other information available, takes simple
- 1 = unbounded problem ($f(\mathbf{x}) <$ `fmin` has occurred: note ∇f is not evaluated in this case)
- 2 = `bl(i)` $>$ `bu(i)` for some `i`
- 3 = infeasible problem detected in Phase 1
- 4 = line search cannot improve f (possibly increase `rgtol`)
- 5 = `mxgr` gradient calls exceeded (this test is only carried out at the start of each iteration)
- 6 = incorrect setting of `m`, `n`, `kmax`, `maxg`, `mlp`, `m0de` or `tol`
- 7 = not enough space in `ws` or `lws`

8 = not enough space in `lp` (increase `mlp`)
9 = dimension of reduced space too large (increase `kmax`)
10 = maximum number of unsuccessful restarts taken
>10 = possible use by later sparse matrix codes

`mxgr` maximum number of gradient calls

`iprint` switch for diagnostic printing (0 = off, 1 = summary, 2 = scalar information, 3 = verbose)

`nout` channel number for output

3. User Subroutines

The user must provide two subroutines as follows

```
subroutine funct(n, x, f, ws, lws, cws)
implicit double precision (a − h, o − z)
dimension x(∗), ws(∗), lws(∗)
character cws(∗)
...
statements to compute f(x) in f from x
...
return
end

subroutine grad(n, x, g, ws, lws, cws)
implicit double precision (a − h, o − z)
dimension x(∗), g(∗), ws(∗), lws(∗)
character cws(∗)
...
statements to compute ∇f(x) in g from x (the user may assume that
a call of grad follows one of funct with the same vector x)
...
return
end
```

The parameters `ws`, `lws` and `cws` in the above subroutines enables data to be passed from the user's calling program to these subroutines.

4. Storage Allocation

User information about the lengths of `ws` and `lws` is supplied to `glcpd` in

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

kk and ll refer to the lengths of ws and lws needed by the user subroutines. kkk and lll are the numbers of locations used by glcpd and are set by glcpd. The rest of ws and lws is used by the files denseL.f or schurQR.f. mxws and mxlws must be set to the total lengths of ws and lws available: a message will be given if more storage is needed.

5. Tolerances, Accuracy and Diagnostics

glcpd uses tolerance and accuracy information stored in

```
common/epsc/eps, tol, emin
common/repc/sgnf, nrep, npiv, nres
common/refactorc/mc, mxmc
common/infoc/rgnorm, vstep, iter, npv, nfn, ngr
```

eps must be set to the machine precision (unit round-off) and tol is a tolerance such that numbers whose absolute value is less than tol are truncated to zero. This tolerance strategy in the code assumes that the problem is well-scaled.

The parameter sgnf is used to measure the maximum allowable relative error in gradient values. If at any stage the accuracy requirement rgtol $<$ sgnf $*$ rgnorm then rgtol is increased to sgnf $*$ rgnorm

The code allows one or more refinement steps after the calculation has terminated, to improve the accuracy of the solution, and a fixed number nrep of such repeats is allowed. However the code terminates without further repeats if no more than npiv pivots are taken. In case of any breakdown, the code is restarted in mode 0. The maximum number of unsuccessful restarts allowed is set in nres.

The basis matrix may be refactorised on occasions, for example to prevent build-up of round-off in the factors or (when using schurQR.f) to limit the growth in the Schur complement. The maximum interval between refactorizations (or size of Schur complement) is set in mxmc.

Default values of all the above parameters are set in block data defaults but can be reset by the user.

infoc returns information about the progress of the method: rgnorm is the norm of the reduced gradient on exit and vstep is the length of the vertical step in the warm start process. iter is the total number of iterations taken, npv is the number of pivots, nfn is the number of function evaluations, and ngr is the number of gradient evaluations.

6. Specifying the Jacobian Matrix A

Two alternative formats are provided for specifying the Jacobian matrix, namely *dense* and *sparse*. The required information must be set in parameters a(*) and

la(*) of glcpd. The code for this is also used in other applications and necessitates provision for a column 0 to be included in A. The data structure must reflect this but no values need be assigned to column 0.

6.1. **Dense format.** In this case A is set in standard Fortran matrix format as a(la,0:m), where la is the 'stride' between columns. la is a single integer which must be greater or equal to n.

In the straightforward case that la $= n$, columns of A follow successively in the space occupied by a(*), starting with column 0.

6.2. **Sparse format.** In this case a and la have dimension a(1:maxa) and la(0:maxla-1), where maxa is at least nnza (the number of nonzero elements in A), and maxla is at least nnza $+ m + 3$. la(0) and 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 (resp.) of all the nonzero elements of A. Entries for each column are grouped together in increasing column order. Within each column group, it is not necessary to have the row indices in increasing order. If appropriate, it can be assumed that all elements of column 0 are zero.

la(0) is a pointer which 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 each of 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 group i of A. Also la(la(0)+m+1)) is set to nnza $+ 1$ (the first unused location in a). Because column 0 of A is zero, the pointers to the start of column groups 0 and 1 are the same.

7. Feasibility determination

The code requires that the lower and upper bounds are consistent ($\mathbf{l} \leq \mathbf{u}$). The method requires iterates which are (primal) feasible. To this end the procedure is divided into three phases. In phase 0, feasibility with respect to the simple bounds on the variables is established. This is always possible, and is immediate in mode 0. In phase 1, a feasible point with respect to all the constraints is sought. This may not always be possible, and the code will terminate with a certificate of 'local infeasibility' (ifail $= 3$). The locally infeasible point is characterised by a subset

of general constraints (J say) that are infeasible, and the sum of infeasibilities of these constraints is minimized, subject to feasibility in the remaining constraints and simple bounds (the set $J^{\perp}$). The set J comprises those constraints i ($i = \texttt{abs(ls(j))}$) for which $\texttt{r(abs(ls(j)))} < 0$, for $\texttt{j} = n+1, \ldots, n+m$. If phase 1 succeeds in finding a feasible point, then phase 2 is entered and the function $f(\mathbf{x})$ is minimized, subject to retaining feasibility.

8. An Illustrative Example

Driver programs are provided to solve the problem with 4 variables and 2 general constraints described by

$$\begin{aligned} &\text{minimize} \quad x_1^{-1} + x_2^{-1} + x_3^{-1} + x_4^{-1} \\ &\text{subject to} \quad \mathbf{l} \leq \begin{bmatrix} \mathbf{x} \\ A^T\mathbf{x} \end{bmatrix} \leq \mathbf{u} \end{aligned}$$

where $l_i = (1/((5-i)10^5)$ and $u_i = 1000$ for $i = 1, \ldots, 4$, $l_5 = l_6 = -\infty$, $u_5 = 0.0401$, $u_6 = 0.010085$ and

$$A^T = \begin{bmatrix} 4 & 2.25 & 1 & 0.25 \\ 0.16 & 0.36 & 0.64 & 0.64 \end{bmatrix}.$$

The problem is derived from the Hock-Schittkowski problem HS72, after making a transformation that linearises the constraints.

The driver program in sparse format is given in the file `hs72.f` and that for dense format in `hs72d.f`.

9. Failure Indications and Troubleshooting

On exit from `glcpd`, the parameter `ifail` indicates the reason for termination. Most of the cases are self evident from the description in Section 2 above. However the case `ifail` = 4 deserves some more detail. In this case the code has failed to find a better (smaller) value of $f(\mathbf{x})$ in a line search. In theory (in the absence of round off errors) this can always be done. Possible reasons are

(1) Mistakes in the formulae for the gradients (or use of finite difference gradients)
(2) Objective function is non-smooth
(3) Too small a tolerance on the gradient asked for (rounding errors in calculating the objective function become dominant).

If 1 and 2 can be ruled out, I find that 3 occasionally happens in problems that I have solved. The current iterate on termination represents the best that can be achieved at the level of precision being used. Unfortunately round-off does limit the accuracy to which problems can be solved.

Another way in which mistakes in the gradient can show up is in slow convergence, leading to `ifail` = 5. This emphasises the importance of getting the gradients

correct. A file `checkg.f` is provided containing a subroutine that checks gradients by finite differences. Users are strongly advised to make use of this before going ahead with `glcpd`. More details are given in `checkg.f` and illustrated in `hs72.f`. Apropos of this, users are advised not to use finite difference approximations to the gradients as an alternative to providing exact formulae. The code is not designed to allow this, and the outcome is unpredictable, as well as being inefficient.

The outcome `ifail` = 10 also merits some comment. The likely cause of this is a failure of the tolerance strategy to recognise zero pivots in the LP-like ratio test. The best advice here is to take every care in scaling the problem. That is, where possible to avoid very large or very small columns in A, and to balance the magnitudes of the different general constraint values, preferably around 1. Likewise, scaling the variables to have values of similar magnitude around 1 can also be beneficial.

Inappropriately large numbers in `x` can also cause difficulty if they arise during the iteration. To prevent this the user is advised to provide realistic bounds on the variables, rather than just using values like `1.D20` when an upper bound is not present.

The file `glcpd.f` contains a subroutine `check` which can carry out checks as the iteration proceeds. There are two calls to `check` from `glcpd` but these are normally commented out. In case of failure, further information as to the cause may be obtained by reinstating these calls.

10. Output

No output occurs if `iprint` = 0. If `iprint` = 1, then one line per iteration is given. Abbreviations used are: in phase 0, `ninfb` is the number of infeasible bounds; in phase 1, `ninf` is the number of infeasible general constraints; in phase 2, `rg` is the reduced gradient norm and `k` is the dimension of the null space. For all phases, `level` is the level of recursion for degeneracy resolution. If `iprint` = 2, then extra scalar information is printed, and if `iprint` = 3, then full details of the iteration are given, but this requires some familiarity with the method to interpret.

11. The Basis Matrix

The code establishes and maintains factors of a nonsingular $n \times n$ 'basis matrix' B. After a column permutation we may write $B = [A\ V]$ where columns of A are gradient vectors of the active constraints, and columns of V are certain unit vectors (columns of the unit matrix corresponding to the free variables). (Here the usage of A is different from that earlier.) The indices of active constraints and free variables are those contained in `ls(j)` for `j` $= 1, \ldots, n$, as described under `ls` in Section 2 of this document. If we define matrices Y and Z by

$$[Y\ Z] = [A\ V]^{-T}$$

then columns of Z are a basis for the null space of the active constraints $\{\mathbf{z} \,|\, A^T\mathbf{z} = \mathbf{0}\}$. The advanced user may wish to carry out matrix operations with Y, Z or their transposes. For example the reduced gradient vector is the vector $Z^T\mathbf{g}$. The means to carry out such operations is provided by the subroutines `fbsub` and `tfbsub` in either `denseL.f` or `schurQR.f` as appropriate, and the user is referred to these files for further information.