M1CG1 – A solver of symmetric linear systems by conjugate gradient iterations, using/building a BFGS/ℓ-BFGS preconditioner

Version 1.2 (October 2011)

J.Ch. Gilbert[†]

1 Goal of the software

The module M1CG1 solves for $x \in \mathbb{R}^n$ the linear system

$$Ax = b, \tag{1.1}$$

by a possibly *preconditioned Fletcher-Reeves conjugate gradient* (CG) algorithm [7, 4, 1, 5]. In this system, the matrix $A \in \mathbb{R}^{n\times n}$ and the vector $b \in \mathbb{R}^n$ are supposed to be known. It is assumed that the matrix A is symmetric. Therefore, solving (1.1) is equivalent to finding a stationary point of the minimization problem

$$\inf_{x\in\mathbb{R}^n} \left(\frac{1}{2}x^{\mathsf{T}}Ax - b^{\mathsf{T}}x\right). \tag{1.2}$$

To be sure to be able to solve the linear system (1.1) [resp. the minimization problem (1.2)] for an arbitrary $b \in \mathbb{R}^n$, the matrix A must be nonsingular [resp. positive

[†]INRIA, B.P. 105, 78153 Le Chesnay Cedex, France. Tel: 33/1/39.63.55.24. E-mail: ‘Jean-Charles.Gilbert@inria.fr’.

definite]. When A is positive definite, (1.1) and (1.2) are equivalent. If the CG algorithm encounters a *quasi-negative curvature direction*, which is a conjugate direction v satisfying

$$v^{\mathsf{T}} A v \leqslant \nu \|v\|^2, \tag{1.3}$$

for some given threshold $\nu \geqslant 0$, M1CG1 just stops. In (1.3) and below, $\|\cdot\|$ denotes the ℓ_2 or *Euclidean norm*, i.e., $\|x\| := (\sum_i x_i^2)^{1/2}$.

The way A is actually stored is irrelevant for M1CG1, since the algorithm requires only products Av of A times various vectors v and since this operation is supposed to be done in a user-supplied subroutine. This gives the possibility to use the code even when A is not known explicitly.

M1CG1 has the nice feature of being able to build itself a preconditioning matrix while it solves the linear system (1.1). This matrix is an approximation of A^{-1} and can then be used to precondition a subsequent linear system with the same matrix A or a matrix close to A, and a different right-hand side. The preconditioning matrix is formed by BFGS updates [3, 1, 5]. It is also possible to form an ℓ-BFGS preconditioning matrix [10, 6, 8], so that the solver is adapted to problems of very large dimension. M1CG1 can also improve a previously built preconditioner; this feature is however presently limited by the fact that the method used to improve the preconditioner (i.e., full BFGS updates or ℓ-BFGS updates with a given selection procedure) must be the same as the used to build the previous preconditioner.

Another feature of M1CG1 is to be able to solve in parallel two linear systems: (1.1) and

$$A\bar{x} = \bar{b}. \tag{1.4}$$

The latter system uses the same matrix A as in (1.1), but another right-hand side $\bar{b}$. The second linear system (1.4) is solved using the conjugate directions generated by the CG iterations for solving (1.1). This means that the two systems are solved without increasing the number of matrix-vector products. This feature is useful for the truncated SQP algorithm in constrained optimization [2].

2 The package

The module M1CG1 includes the following subroutines:

– `m1cg1`: the main subroutine, which has to be called and that calls `m1cg1a` after having structured the available memory;
– `m1cg1a`: the actual solver;
– `dbfgsi`, `dysave`, `dbfgsl`, `deuclid`, `dsyev`, `l2bfgs`: subroutines used to build a BFGS preconditioning matrix.

On his part, the user has to provide:

– a subroutine, called `mvprod` by M1CG1, which computes matrix-vector products Av for arbitrary vectors $v \in \mathbb{R}^n$; the method of storage for the matrix is left to the user convenience.

3 Usage

3.1 The optimization subroutine `m1cg1`

The subroutine definition statement is the following.

```
subroutine m1cg1 (mvprod, n, x, b, a, xx, bb, two, epsneg,
                  restol, absrel, iter, imp, io, imode,
                  mode, w, nw, bfgsp, pmat0, npmat0, m0,
                  ilm0, nilm0, wlm0, nwlm0, bfgsb, pmat1,
                  npmat1, m1, ilm1, nilm1, wlm1, nwlm1,
                  select, izs, rzs, dzs)
```

In the description that follows, an argument flagged with (I) means that it is an *input* variable, which has to be initialized before calling `m1cg1`, an argument flagged with (O) means that it is an *output* variable, which has a meaning on return from `m1cg1` only, and an argument flagged with (IO) is an *input-output* argument, which has to be initialized and has a meaning after the call to `m1cg1`. Arguments of the type (O) and (IO) are usually modified in `m1cg1`. Therefore, they *should not be* `fortran` *constants*!

`mvprod`: Generic name of the user-supplied subroutine (see Section 3.2). This name must be declared `external` in the program calling `m1cg1`.

`n` (I): `integer` variable. Dimension n of the problem. It must be $\geqslant 1$.

`x` (IO): `double precision` array of dimension n.

On entry: the initial guess for the solution x of (1.1).

On return: the approximate solution of (1.1) computed by M1CG1.

If M1CG1 stops without doing any iteration because a quasi-negative curvature has been encountered, `x` contains $(Ax_1 - b)$. If M1CG1 does more than one iteration, `x` contains the last computed value of x.

`b` (I): `double precision` array of dimension n. Right hand side b of the linear system (1.1).

`a` (I): `double precision` array. The matrix A in (1.1). How this array is filled up is left to the user, as well as the dimension(s) of `a`. M1CG1 just passes the address to the user-supplied subroutine that is named `mvprod` inside M1CG1.

`xx` (IO): `double precision` array of dimension n (see the argument `two`).

On entry: the initial guess for the solution $\bar{x}$ of (1.4).

On return: the approximate solution of (1.4) computed by M1CG1.

`bb` (I): `double precision` array of dimension n. Right hand side $\bar{b}$ of the linear system (1.4) (see the argument `two`).

`two` (I): `logical` variable. If `.false.`, only (1.1) is solved and the variables `xx` and `bb` are ignored. If `.true.`, (1.4) is also solved, and this is done by the *conjugate direction algorithm*, using the conjugate directions generated by the CG algorithm solving (1.1).

`epsneg` (IO): `double precision` variable.

On entry: Contributes to the setting of the value of ν in (1.3). The precise meaning of `epsneg` depends on the value of `imode(3)` (another parameter). It must be nonnegative. If during the CG iterations, condition (1.3) is satisfied with v as the current conjugate direction, the algorithm stops, since the matrix A is not considered to be positive definite.

On return: Value of the last Rayleigh quotient $(v^{\mathsf{T}}Av)/\|v\|^2$ encountered (unchanged if the conjugate direction v is zero).

`restol` (IO): `double precision` variable.

On entry: It must be nonnegative. Its interpretation depends on the flag `absrel` (see below).

- If `absrel` $= 0$, `restol` is assumed to give the required precision on the *absolute residual*. In other words, if during the CG iterations, the absolute residual satisfies
$$\|Ax - b\| \leqslant \texttt{restol},$$
the algorithm stops, since (1.1) is considered to be solved by the current x with enough precision.
- If `absrel` $= 1$, `restol` is assumed to give the required precision on the *relative residual*. In other words, if during the CG iterations, the relative residual satisfies
$$\frac{\|Ax - b\|}{\|Ax_1 - b\|} \leqslant \texttt{restol},$$
the algorithm stops, since (1.1) is considered to be solved by the current x with enough precision.

When `two` = `.true.`, there is no stopping test based on the behavior of the conjugate direction algorithm on (1.4). If (1.4) is solved before finding the approximate solution of (1.1), $\bar{x}$ is modified by adding terms that are zero in exact arithmetic and that should be close to zero on a computer.

On return: when M1CG1 stops with `mode` $= 0$,
$$\texttt{restol} = \begin{cases} \|Ax - b\| & \text{if } \texttt{absrel} = 0 \\ \frac{\|Ax-b\|}{\|Ax_1-b\|} & \text{if } \texttt{absrel} = 1, \end{cases}$$
where x is the final iterate.

`absrel` (I): `integer` variable, which can take the value `0` or `1`. It specifies how M1CG1 has to understand the value of the argument `restol`. See the description of that argument for the role of `absrel`.

`iter` (IO): `integer` variable.

On entry: maximal number of iterations authorized.

On return: actuel number of iterations performed

During the run, `iter` is used as iteration counter, so that the user of M1CG1 can know the iteration number each time `mvprod` is called. It is important, however, not to change the value of `iter` while M1CG1 is running.

`imp` (I): `Integer` variable that controls the printings via the I/O channel `io` (it is another argument).

$= 0$: nothing is printed.

$\geqslant 1$: printings on entry and return; M1CG1 prints also error messages.

$\geqslant 2$: one line is also printed at each CG iteration. It gives

- `iter`: iteration counter;
- `cost`: the cost $\frac{1}{2}x^{\mathsf{T}}Ax - b^{\mathsf{T}}x$;
- `|r|` or `|r|/|r1|`: the absolute (if `absrel` $= 0$) or relative (if `absrel` $=$ 1) norm of the residual (controlled by `restol`); does not decrease monotonically;
- `<(r,Pr)`: angle between the residual $r = Ax - b$ and the preconditioned residual Pr; should be 0 if `bfgsp` $= 0$;
- `(Av,v)/|v|^2`: Rayleigh quotient of A along the current conjugate direction v;
- `conj`: cosine of the angle between 2 succesive residuals (for the scalar product defined by the preconditioning matrix P); should be very close to 0;
- `alpha`: step-size; always positive;
- `|x|`: norm of the current approxinate solution x; should increase if `bfgsp` $= 0$;
- `S`: selection flag (only present if an ℓ-BFGS preconditioner is built, i.e., if `bfgsb` $= 2$); '`S`' means that the (Av, v)-pair has been selected, '`-`' means that it has not been selected.

If `two` $=$ `.true.`, `cost` and (`|r|` or `|r|/|r1|`) is also printed for the linear system (1.4).

`io` (I): `integer` variable. Output channel for printings.

`imode` (I): `integer` variable. Input mode of M1CG1.

`imode(1)`:

- If `imode(1)` $= 0$, the value of `x` is set to 0 by M1CG1 and the CG algorithm starts with $x_1 = 0$;
- If `imode(1)` $= 1$, the value in `x` is taken as starting point x_1 by the CG algorithm.

`imode(2)`:

- If `imode(2)` $= 0$, the value of `xx` is set to 0 by M1CG1 and the conjugate direction algorithm starts with $\bar{x}_1 = 0$;
- If `imode(2)` $= 1$, the value in `xx` is taken as starting point $\bar{x}_1$ by the conjugate direction algorithm.

`imode(3)`: Specifies the meaning of the parameter `epsneg`.

- If `imode(3)` $= 0$, `epsneg` is considered to be the value of ν in (1.3).

- If imode(3) $= 1$, epsneg is considered to be the smallest authorized value for the ratio between the minimum and the maximum Rayleigh quotient encountered during the CG iterations. In other words, the inverse of epsneg is the maximum condition number that the CG iterations are supposed to support. In fact, ν in (1.3) is set to epsneg multiplied by the maximum Rayleigh quotient encountered during the CG iterations. In this case, ν depends on the iteration counter.

mode (0): integer variable. Output mode of M1CG1.

$= 0$: normal terminaison (stop on restol);
$= 1$: a dimension argument has a wrong value;
$= 2$: (not used);
$= 3$: A is probably indefinite (Rayleigh quotient less than ν);
$= 4$: maximum number of iterations has been reached;
$= 5$: (not used);
$= 6$: the initial residual is zero; hence x_1 is solution of (1.1);
$= 7$: error in the subroutine dysave, this may be due to a wrong value of select, a scalar product $y^\mathsf{T} s < 0$ (rounding error?), or a negative value of m1,
$= 8$: significant increase of the cost, due to rounding errors.

w (I): double precision array of dimension nw. Working zone.

nw (I): integer variable. Give the dimension of w. One must have:

$$\begin{aligned} \texttt{nw} &\geqslant 5n + \texttt{m0} \quad \text{if } \texttt{two} = \texttt{.false.} \\ \texttt{nw} &\geqslant 6n + \texttt{m0} \quad \text{if } \texttt{two} = \texttt{.true.} \end{aligned}$$

bfgsp (I): integer variable. It specifies what type of preconditioning information is made available to M1CG1 to precondition the current CG run.

$= 0$: no preconditioning information;
$= 1$: a full symmetric positive definite preconditioning $n \times n$ matrix is available in the structure (pmat0, npmat0);
$= 2$: a limited memory BFGS preconditioning matrix is available in the ℓ-BFGS structure (m0, ilm0, nilm0, wlm0, nwlm0); this structure can be built up by a preceding call to m1cg1.

pmat0, npmat0 (I): If bfgsp $= 1$, this structure must contain preconditioning information. It is not used otherwise.

pmat0: is a double precision array of dimension npmat0. It must contain an $n \times n$ symmetric positive definite matrix P approximating the inverse of A. The (i, j)th element P_{ij} of P is supposed to be in pmat0(i,j), so that the leading dimension of pmat0 must be $\geqslant$ n.

npmat0: integer variable. It is the dimension of pmat0 and must be $\geqslant n^2$.

m0, ilm0, nilm0, wlm0, nwlm0 (I): If bfgsp $= 2$, this structure must contain ℓ-BFGS preconditioning information. It is not used otherwise.

m0: integer variable. It is the number of (Av, v)-pairs stored in the structure.

ilm0: integer array of dimension nilm0.

`nilm0`: `integer` variable, giving the dimension of `ilm0`. There must hold

$$\texttt{nilm0} \geqslant \texttt{m0} + 4.$$

`wlm0`: `double precision` array of dimension `nwlm0` that contains the (Av, v)-pairs.

`nwlm0`: `integer` variable, giving the dimension of `wlm0`. There must hold

$$\texttt{nwlm0} \geqslant \texttt{m0}(2n{+}1) + 1.$$

`bfgsb` (I): `integer` variable. It specifies the type of BFGS preconditioning matrix that M1CG1 has to build while it solves (1.1).

= 0: M1CG1 will not build any preconditioning matrix.

= 1: M1CG1 has to build a full BFGS preconditioning matrix in the BFGS structure (`pmat1`, `npmat1`). If

$$\texttt{pmat1}(1,1) \leqslant 0,$$

the updates start from scratch, otherwise `pmat1` is updated.

= 2: M1CG1 has to build a limited memory BFGS preconditioning matrix in the ℓ-BFGS structure (`m1`, `ilm1`, `nilm1`, `wlm1`, `nwlm1`). If one of the following two conditions

$$\texttt{ilm1(1)} \leqslant 0,$$
$$\texttt{ilm1(1)} > 0 \qquad \text{and} \qquad \texttt{select} = 1,$$

is satified, the updates start from scratch, otherwise the ℓ-BFGS structure (`m1`, `ilm1`, `nilm1`, `wlm1`, `nwlm1`) is updated, provided the selection procedure specified by the argument `select` is the same as the one that was used to built that previous preconditioner.

`pmat1`, `npmat1` (O): This structure is filled in if `bfgsb` = 1 and not touched otherwise. Same meaning as for the structure (`pmat0`, `npmat0`).

`m1`, `ilm1`, `nilm1`, `wlm1`, `nwlm1` (O): This structure is filled in if `bfgsb` = 2 and not touched otherwise. Same meaning as for the structure (`m0`, `ilm0`, `nilm0`, `wlm0`, `nwlm0`).

`select` (I): `integer` variable. In the case when `bfgsb` = 2, this argument monitors the selection of the (Av, v)-pairs used to build the ℓ-BFGS preconditioning matrix. The meaning is as follows.

= 0: *FIFO (First In First Out) selection*: the oldest pair is discarded and the new one is saved.

= 1: *Mexican selection* [9]: the pairs are distributed uniformly according to the iteration counter of the current CG run.

= 2: *Rayleigh quotient selection*: it is tried to have pairs with a Rayleigh quotient distributed as uniformly as possible on the logarithmic scale.

In this version of the software, if it is desired to improve a previously built preconditioner, stored in the structure (`m1`, `ilm1`, `nilm1`, `wlm1`, `nwlm1`), the selection procedure specified by `select` must be the same as the one that was used to build that previous preconditioner.

`izs`, `rzs`, `dzs`: Variables or arrays, whose type is respectively `integer`, `real`, and `double precision`. These variables/arrays are not used by M1CG1. They are just passed to the user-supplied subroutine `mvprod`, so that the user can use them as wished.

3.2 The user-supplied subroutine `mvprod`

`Mvprod` is the user-supplied subroutine that computes the matrix-vector products Av for arbitrary vectors v, which are actually conjugate directions. M1CG1 assumes that this subroutine is defined by the following statement.

```
subroutine mvprod (n, a, v, av, izs, rzs, dzs)
```

Meaning of the arguments:

`n`, `a`, `izs`, `rzs`, `dzs`: same meaning as the arguments of `m1cg1` with the same name.

`v` (I): `double precision` array of dimension n. A vector $v \in \mathbb{R}^n$ given to multiply the matrix A.

`av` (O): `double precision` array of dimension n. Vecteur Av, product of the matrix A by the vector v, that `mvprod` has to compute.

4 Implementation details

4.1 Calling sequence

Necessarily, the instructions preceding the call to `m1cg1` must include the following items:

1. the declaration in `external` of the name of the user-supplied subroutine doing matrix-vector products (named `mvprod` by M1CG1);
2. the calculation of the starting point x_1 (and possibly $\bar{x}_1$);
3. the initialization of the input arguments of `m1cg1`;
4. the call to `m1cg1`.

4.2 Storage of the ℓ-BFGS matrices

The ℓ-BFGS matrices are stored in the structures formed of the variables `mX`, `ilmX`, `nilmX`, `wlmX`, and `nwlmX`, where the `X` stands for `0` (preconditioning matrix) or `1` (constructed preconditioner). A few has been said about these variables in the description of the subroutine m1cg1. Here are more details.

`mX`: `integer` variable. It is the maximal number of (Av, v)-pairs that can be stored in the structure.

`ilmX(1)`: `integer` variable. It is the number of (Av, v)-pairs that have been selected for storage in the ℓ-BFGS structure, possibly replacing older pairs.

`ilmX(2)`: `integer` variable, specifying the selection procedure that has been used to select the (Av, v)-pairs stored in the ℓ-BFGS structure; see the variable `select`.

`ilmX(3)`: `integer` variable. It is a cyclic pointer to the oldest (Av, v)-pair stored in the structure. Ones it has been initialized, its value is between `1` and `mX`. The word *cyclic* means here that, when its value is `mX` and it is incremented, its new value is `1`.

`ilmX(4)`: `integer` variable. It is a cyclic pointer to the newest (Av, v)-pair stored in the structure. Ones it has been initialized, its value is between `1` and `mX`.

`ilmX(4+i)`, for `i` $= 1$ to `mX`: `integer` variables; `ilmX(4+i)` is the index of the $(Av,$ $v)$ pair with the `i`th smallest Oren-Luenberger factor.

`nilmX`: `integer` variable; dimension of `ilmX`; $=$ `mX` $+ 3$.

`wlmX(1)`: `double precision` variable. It is the first element of an array of dimension `(n,mX)`, containing the Av part of the `mX` (Av, v)-pairs.

`wlmX(n*mX+1)`: `double precision` variable. It is the first element of an array of dimension `(n,mX)`, containing the v part of the `mX` (Av, v)-pairs.

`wlmX(2*n*mX+1)`: `double precision` variable. It is a scalar preconditioner.

`wlmX(2*n*mX+2)`: `double precision` variable. It is the first element of an array of dimension `mX`, containing the Oren-Luenberger factors of the (Av, v)-pairs.

`nwlmX`: `integer` variable; dimension of `wlmX`; it must be $\geqslant$ `mX*(2*n+1)+1`.

5 Numerical results

5.1 Performance of the BFGS preconditioners

This section aims at experimenting the performance of the BFGS preconditioner built by M1CG1. The experiments are done for solving a randomly generated linear system (1.1) of order $n = 100$ with a positive definite matrix A and a right hand side b. The experiments are implemented as test 10 in the program `quad.f` located in the directory `examples` of the standard distribution. It can be executed by typing

```
make quad
quad 10
```

which generates Matlab codes that can be run to obtain the curves in figure 5.1 below.

In the first experiment, a BFGS preconditioner is formed during a single run made of $n_k = 50k$ $(k = 0, 1, \ldots, 10)$ unpreconditioned CG iterations. This preconditioner is then used to solve the same linear system with preconditioned CG iterations. The number of iterations to get an absolute precision of 10^{-10} (`restol`=`1.d-10` and `absrel`=`0`).

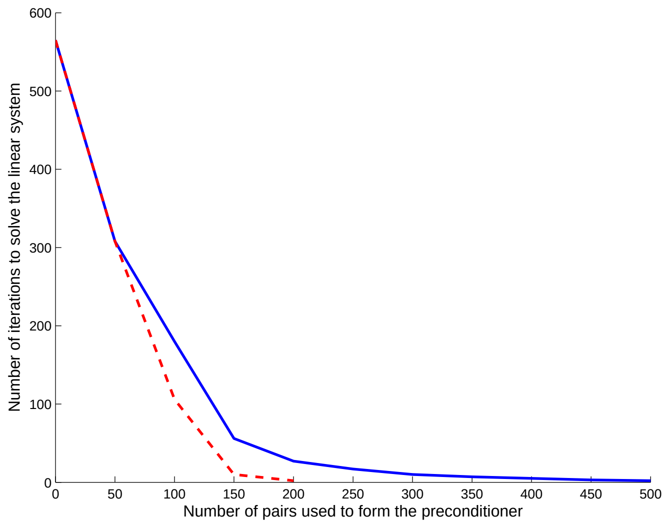

Figure 5.1: Number of iterations needed to solve a randomly generated 100×100 linear system with a BFGS preconditioner that was previously built using the number n_k of (Av, v)-pairs given in abscissa. The solid blue curve refers to the first experiment: the preconditioner is generated by a single CG run. The dashed red curve refers to the second experiment: the preconditioner is generated/improved by preconditioned CG runs of 50 iterations.

is shown in figure 5.1 as a function of n_k (solid blue curve): there is a clear decrease of the number of preconditioned CG iterations when n_k increases. Observe, however that for $n_k = 100$ (the order of the linear system), the number of preconditioned CG iterations to solve the linear system is not 1, but still 180. This is not in concordance with the theory (see [5] for example) and, by lack of a deeper analysis, we attribute this to rounding errors (to back this opinion, note that a solution to the linear system by the unpreconditioned CG algorithm requires here 565 iterations to reach the required precision, and not 100 as claimed by the theory, which assumes that the computation is done in exact arithmetics).

In the second experiment, a BFGS preconditioner is also formed using $n_k = 50k$ ($k = 0, 1, \dots, 4$) CG iterations, but these iterations are not taken from a single unpreconditioned CG run, but out of k preconditioned CG runs, each of which being formed of 50 iterations. To be more precise, a first preconditioner is obtained using the first 50 (Av, v)-pairs generated by an unpreconditioned CG run. Next, the preconditioner is improved by the first 50 (Av, v)-pairs generated by a CG run preconditioned by the matrix obtained after the first run. And so on, k times. The final preconditioner is then used to solve the same linear system. The required number of iterations as a function of n_k is shown by the dashed red curve in figure 5.1. We quote two facts: as for the solid blue curve, there is a clear decrease of the number of preconditioned CG iterations when n_k increases; in addition the number of iterations is significantly less than the one obtained in the first experiment with the same n_k.

5.2 Performance of the ℓ-BFGS preconditioners

This section aims at experimenting the performance of the ℓ-BFGS preconditioners built by M1CG1, those obtained when `bfgsb` is set to `2` and `select` is set to `0`, `1`, or `2`. The experiments are done for solving the same randomly generated linear system (1.1) as in section 5.1, in particular, the order of the system is $n = 100$. The experiments are implemented as tests 11, 12, and 13 in the program `quad.f` located in the directory `examples` of the standard distribution. They can be run by typing

```
make quad
quad 11
quad 12
quad 13
```

which generates Matlab codes that can be launched to obtain the curves in the figures 5.2, 5.3, and 5.4 below.

In the first experiment, we compare the performance of the BFGS and ℓ-BFGS preconditioners, when these are formed using $m = 10k$ updates (for $k = 0, 1, \ldots, 10$). In this experiment, the ℓ-BFGS preconditioner is allowed to store 100 (Av, v)-pairs, so that the pair selection option, monitored by `select`, is ineffective. The comparison is illustrated in figure 5.2. A first observation is that the ℓ-BFGS preconditioner

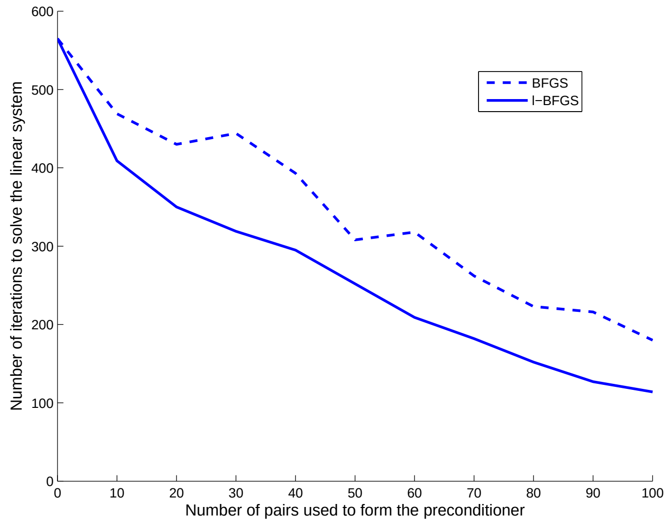

Figure 5.2: Number of iterations needed to solve a randomly generated 100×100 linear system with a BFGS (dashed blue curve) or ℓ-BFGS (solid blue curve) preconditioner that was previously built using the number of (Av, v)-pairs given in abscissa.

(solid blue curve) improves significantly when the number of updates increases; this observation was already observed in section 5.1 for the BFGS preconditioner, but the dashed blue curve shows here that this decrease is not monotone. A second observation is that the ℓ-BFGS preconditioner performs significantly better than the BFGS preconditioner. We attribute this to the diagonal scaling included in the ℓ-BFGS preconditioner, which can be rescaled at each update, but not in the BFGS preconditioner. Note, however, that when m is large, the use of the ℓ-BFGS preconditioner may require more computations than the BFGS preconditioner.

In the second experiment, we investigate the performance of the 3 possible selection procedures, corresponding to the values `0`, `1`, or `2` of the argument `select`. For this, we fix to $m = 20$ the number of (Av, v)-pairs that the ℓ-BFGS preconditioner can store and let M1CG1 select the pairs on an unpreconditioned CG run made of $n_k = mk$ iterations, for $k = 1, \ldots, 100/m$, which is the abscissa of figure 5.3 below. The ordinate in this

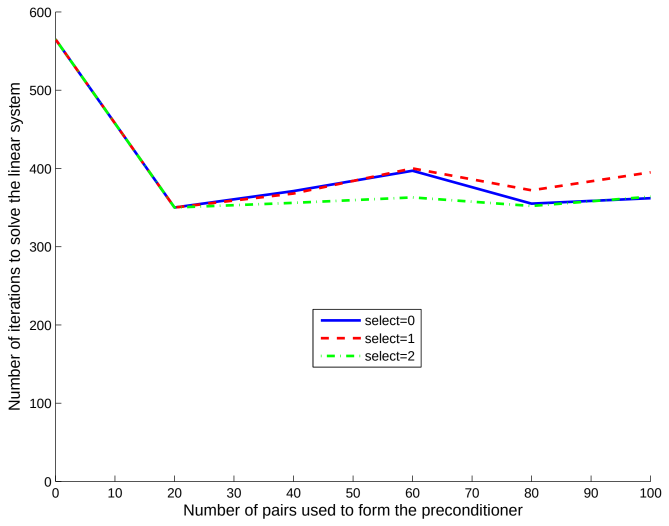

Figure 5.3: Number of iterations needed to solve a randomly generated 100×100 linear system with an ℓ-BFGS preconditioner, made of $m = 20$ (Av, v)-pairs, previously built by selecting these pairs on the n_k iterations given in abscissa. The curves correspond to a different selection procedure: `select` $=$ `0` for the solid blue curve, `select` $=$ `1` for the dashed red curve, and `select` $=$ `2` for the dotted-dashed green curve.

figure gives the number of iterations required by M1CG1 using the built preconditioner, to solve the linear system to the absolute precision 10^{-10}. The results are disappointing, showing that the selection procedures are inefficient in selecting the appropriate pairs. Among the selection procedures proposed by M1CG1, the one based on the Rayleigh quotient (`select` $=$ `2`), looks better than the others. Setting the ℓ-BFGS size to $m = 5$ or $m = 20$ provides similar results.

In the third and last experiment, we want to test the ability of M1CG1 to improve an ℓ-BFGS preconditioner by incorporating new (Av, v)-pairs during successive CG runs. For this, an ℓ-BFGS preconditioner is built during $m = 20$ unpreconditioned CG iterations. Then this preconditioner is updated $k-1$ times ($k = 1, \ldots, 5$), each time by m preconditioned CG iterations (the preconditioner being the one so obtained before the current run). Finally, the linear system is solved with the resulting preconditioner. Figure 5.4 shows the two curves obtained with the selection procedure flag `select` $=$ `0` or `select` $=$ `2` (there is no curve corresponding to `select` $=$ `1`, since the principle

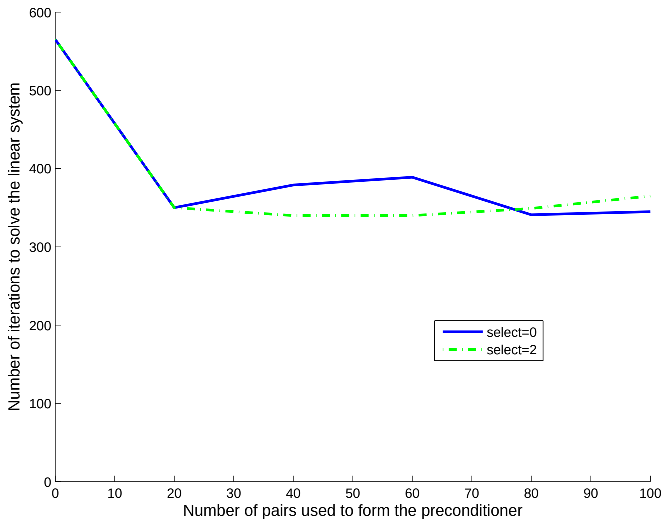

Figure 5.4: Number of iterations needed to solve a randomly generated 100×100 linear system with an ℓ-BFGS preconditioner, made of $m = 20$ (Av, v)-pairs, previously built by selecting these pairs during 5 runs made of 20 preconditioned CG iterations. The curves correspond to a different selection procedure: `select` $= 0$ for the solid blue curve and `select` $= 2$ for the dotted-dashed green curve.

underlying the Mexican selection procedure is not adapted to this experiment). In ordinate, one finds the number of iterations required to solve the linear system with the preconditioner built with mk updates ($k = 0, \ldots, 5$), a number given in abscissa.

These last two experiments clearly indicate that more research is needed to improve the selection procedures. In the present development of the software, if it is desired to improve the ℓ-BFGS preconditioner, it is much better to increase its number m of (Av, v)-pairs (provided the memory space allows such an increase) than to make it collecting information on a number of CG iterations larger than m.

References

[1] J.F. Bonnans, J.Ch. Gilbert, C. Lemaréchal, C. Sagastizábal (2006). *Numerical Optimization – Theoretical and Practical Aspects* (second edition). Universitext. Springer Verlag, Berlin. [authors] [editor] [google books]. 1, 2

[2] L. Chauvier, A. Fuduli, J.Ch. Gilbert (2003). A truncated SQP algorithm for solving nonconvex equality constrained optimization problems. In G. Di Pillo, A. Murli (editors), *High Performance Algorithms and Software for Nonlinear Optimization*, pages 146–173. Kluwer Academic Publishers, Dordrecht. 2

[3] J.E. Dennis, R.B. Schnabel (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs. 2

[4] R. Fletcher, C.M. Reeves (1964). Function minimization by conjugate gradients. *The Computer Journal*, 7, 149–154. 1

[5] J.Ch. Gilbert (2011). *Éléments d'Optimisation Différentiable – Théorie et Algorithmes*. Syllabus de cours à l'ENSTA, Paris. [page internet]. 1, 2, 10

[6] J.Ch. Gilbert, C. Lemaréchal (1989). Some numerical experiments with variable-storage quasi-Newton algorithms. *Mathematical Programming*, 45, 407–435. 2

[7] M.R. Hestenes, E. Stiefel (1952). Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49, 409–436. 1

[8] D.C. Liu, J. Nocedal (1989). On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45, 503–520. 2

[9] J.L. Morales, J. Nocedal (2000). Automatic preconditioning by limited memory quasi-Newton updating. *SIAM Journal on Optimization*, 10, 1079–1096. 7

[10] J. Nocedal (1980). Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35, 773–782. 2

Index