Oqla user's guide

Jean Charles Gilbert, Émilie Joannopoulos

Oqla user's guide

Jean Charles Gilbert[1] and Émilie Joannopoulos[1]

January 28, 2015

[1]INRIA-Paris-Rocquencourt, BP 105, F-78153 Le Chesnay Cedex (France); e-mails: `Jean-Charles.Gilbert@inria.fr` and `emilie.joannopoulos@inria.fr`.

1 Presentation

1.1 The problem to solve

The piece of software `Oqla` has been designed to solve a *convex quadratic optimization problem*. This one is assumed to have the following form

$$\text{(QP)} \quad \begin{cases} \inf_x \ g^\mathsf{T} x + \frac{1}{2} x^\mathsf{T} H x \\ l_B \leqslant x \leqslant u_B \\ l_I \leqslant A_I x \leqslant u_I \\ A_E x = b_E, \end{cases} \tag{1.1}$$

where

- $g \in \mathbb{R}^n$ is the gradient of the *objective* at the origin; n is a nonzero integer;
- $H \in \mathbb{R}^{n \times n}$ is the Hessian of the objective, which is assumed to be symmetric and positive semi-definite;
- $l_B \in \overline{\mathbb{R}}^n$ and $u_B \in \overline{\mathbb{R}}^n$ specify lower and upper bounds on x; they must satisfy $l_B < u_B$ (i.e., $l_i < u_i$ for all indices $i \in [1\,{:}\,n] \equiv \{1, \ldots, n\}$);
- $A_I \in \mathbb{R}^{m_I \times n}$ is the inequality constraint Jacobian; I is supposed to be a finite index set whose cardinality is $m_I := |I|$; the rows of A_I are denoted by A_i for $i \in I$;
- $l_I \in \overline{\mathbb{R}}^{m_I}$ and $u_I \in \overline{\mathbb{R}}^{m_I}$ specify lower and upper bounds on $A_I x$; they must satisfy $l_I < u_I$ (i.e., $l_i < u_i$ for all indices $i \in I$);
- $A_E \in \mathbb{R}^{m_E \times n}$ is the equality constraint Jacobian; E is supposed to be a finite index set whose cardinality is $m_E := |E|$; the rows of A_E are also denoted by A_i for $i \in E$;
- $b_E \in \mathbb{R}^{m_E}$.

Since H may vanish, Oqla can solve a *linear optimization* problem.

We will denote the objective by $q : \mathbb{R}^n \to \mathbb{R}$, hence with the value at $x \in \mathbb{R}^n$ given by

$$q(x) = g^\mathsf{T} x + \frac{1}{2} x^\mathsf{T} H x.$$

Below, the inequality constraints will be written more compactly as follows $l \leqslant (x, A_I x) \leqslant u$, in which

$$l = \begin{pmatrix} l_B \\ l_I \end{pmatrix} \qquad \text{and} \qquad u = \begin{pmatrix} u_B \\ u_I \end{pmatrix}.$$

It is convenient to denote by $A_B = I_n$ the identity matrix of order n. Then the inequalities $l \leqslant (x, A_I x) \leqslant u$ read in a similar manner $l_B \leqslant A_B x \leqslant u_B$ and $l_I \leqslant A_I x \leqslant u_I$. We also take the notation

$$m = n + m_I + m_E \qquad \text{and} \qquad A = \begin{pmatrix} A_B \\ A_I \\ A_E \end{pmatrix} \in \mathbb{R}^{m \times n}.$$

When the optimal value of (1.1) is $-\infty$ the problem is said to be *unbounded*; this means that there is a sequence $\{x_k\}$ satisfying the constraints such that $q(x_k) \to -\infty$. By convention, the optimal value is $+\infty$ when the problem is *infeasible*, which means that there is no x satisfying the constraints (i.e., the constraints are not ompatible). By a result of Frank and Wolfe [12; 1956, appendix (i)] (also valid, actually, when the quadratic problem is nonconvex), in all the other cases, problem (1.1) has a solution.

An inequality constraint with the index $i \in B \cup I$ is said to be *active* at x if $A_i x = l_i$ or u_i. The *active set* (AS) at x is defined by

$$\mathcal{A}(x) := \{i \in B \cup I : A_i x = l_i \text{ or } u_i\}.$$

The *Lagrangian* of problem (1.1) is the function

$$(x, \tilde{\lambda}) \mapsto \ell(x, \tilde{\lambda}) \in \mathbb{R}$$

defined at $(x, \tilde{\lambda}) \in \mathbb{R}^n \times \mathbb{R}^{(n+m_I)+(n+m_I)+m_E}$ by

$$\begin{aligned} \ell(x, \tilde{\lambda}) = g^T x + \frac{1}{2} x^T H x + (\tilde{\lambda}^l_{B\cup I})^T (l - A_{B\cup I} x) \\ + (\tilde{\lambda}^u_{B\cup I})^T (A_{B\cup I} x - u) \\ + \tilde{\lambda}_E^T (A_E x - b_E), \end{aligned} \tag{1.2}$$

in which the vector of *multipliers* or *dual variables* is

$$\tilde{\lambda} := (\tilde{\lambda}^l_{B\cup I}, \tilde{\lambda}^u_{B\cup I}, \tilde{\lambda}_E) \in \mathbb{R}^{n+m_I} \times \mathbb{R}^{n+m_I} \times \mathbb{R}^{m_E}.$$

1.2 The closest feasible problem

Since the constraints of the quadratic problem (1.1) are linear, there is a smallest vector $\bar{s} = (\bar{s}_I, \bar{s}_E) \in \mathbb{R}^{m_E} \times \mathbb{R}^{m_I}$ such that the shifted constraints

$$l_B \leqslant x \leqslant u_B, \qquad l_I \leqslant A_I x + \bar{s}_I \leqslant u_I, \qquad \text{and} \qquad A_E x + \bar{s}_E = b_E$$

are feasible for x [6, 14]. In this claim, smallness is measured by the Euclidean norm. This vector $\bar{s}$ is called the *smallest feasible shift*. Of course $\bar{s} = 0$ if and only if the constraints of problem (1.1) are feasible.

The quadratic problem obtained from (1.1) by shifting its constraints with $\bar{s}$ is called the *closest feasible problem*. It reads

$$(\mathrm{QP}_{\bar{s}}) \quad \begin{cases} \inf_x \ q(x) \\ l_B \leqslant x \leqslant u_B \\ l_I \leqslant A_I x + \bar{s}_I \leqslant u_I \\ A_E x + \bar{s}_E = b_E. \end{cases} \tag{1.3}$$

By definitioni of $\bar{s}$, this problem is always feasible, but it may be unbounded. Oqla always solves this problem, when it has a solution. Of course, when the original problem is feasible (i.e., $\bar{s} = 0$) and bounded, Oqla solves the original problem. When problem (1.1) is infeasible, computing a solution to the closest feasible problem (1.3) can be interesting when the quadratic problem is generated by the SQP algorithm [4, 19].

When (1.3) has no solution (i.e., it is unbounded), Oqla computes a *direction of unboundedness* of the problem, that is a direction $d \in \mathbb{R}^n$ such that

$$g^T d < 0, \quad Hd = 0, \quad d \in [l_B, u_B]^\infty, \quad A_I d \in [l_I, u_I]^\infty, \quad \text{and} \quad A_E d = 0. \tag{1.4}$$

We have denoted by C^∞ the asymptotic/recession cone of a convex set C [23, 18, 3]. The last three conditions express the fact that d is an asymtotic direction of the feasible set of (1.3). This direction of unboundedness d has therefore the property that, if x is feasible for (1.3), $x + \alpha d$ is also feasible for (1.3) whatever is $\alpha \geqslant 0$, and $q(x + \alpha d) \to -\infty$ when $\alpha \to +\infty$. When the closest feasible problem (1.3) is infeasible, computing a direction of unboundedness can be interesting when the quadratic problem is generated by the SQP algorithm [4, 19].

1.3 The solution method

1.3.1 Algorithmic scheme

The method used by `Oqla` to solve problem (1.1) is the *augmented Lagrangian algorithm* or *method of multipliers*. This algorithm was introduced for equality constrained nonlinear optimization problems by Hestenes and Powell [16, 22], and extended to inequality constrained problems by Rockafellar, Buys, Arrow, Gould, and Howe [24, 5, 25, 2, 26]. Its global linear convergence has been analyzed in [7, 6] (more references on the algorithm are given in the latter papers).

The method works on a transformed version of the original problem (1.1), obtained by using an auxiliary variable $y \in \mathbb{R}^{m_I}$, namely

$$\text{(QP')} \quad \begin{cases} \inf_{(x,y)} \ q(x) \\ l \leqslant (x,y) \leqslant u \\ A_I x = y \\ A_E x = b_E \end{cases} \tag{1.5}$$

Note that the auxiliary variable y is only associated with the complex linear inequalities $l_I \leqslant A_I x \leqslant u_I$, not with the bound constraints $l_B \leqslant x \leqslant u_B$. The goal is to have simple bound constraints on the variables, which are dealt with explicitely by the algorithm (not relaxed), and additional linear equality constraints.

The *augmented Lagrangian* (AL), which plays a major part in the solution method, is the function defined at $(x, y, \lambda) \in \mathbb{R}^n \times \mathbb{R}^{m_I} \times \mathbb{R}^{m_I+m_E}$ by

$$\begin{aligned} \ell_r(x,y,\lambda) = g^\mathsf{T} x + \frac{1}{2} x^\mathsf{T} H x + \lambda_I^\mathsf{T}(A_I x - y) + \frac{r}{2}\|A_I x - y\|^2 \\ + \lambda_E^\mathsf{T}(A_E x - b_E) + \frac{r}{2}\|A_E x - b_E\|^2, \end{aligned} \tag{1.6}$$

where $r > 0$ is called the *augmentation parameter* and $\|\cdot\|$ denotes the Euclidean norm. It is not deficult to see that problem (QP') can also be written

$$\inf_{l \leqslant (x,y) \leqslant u} \ \sup_{\lambda \in \mathbb{R}^{m_I+m_E}} \ \ell_r(x,y,\lambda), \tag{1.7}$$

so that the AL can be viewed as a function "dualizing" (or relaxing) the equality constraints of (QP'), while the bound constraints of that problem are kept unchanged. Note that the multiplier λ used here is different form the multiplier $\tilde{\lambda}$ used in (1.2), although there is a link betwenn them. The dual problem, associated with the expression (1.7) is obtained by inverting the infimum and supremum:

$$\sup_{\lambda \in \mathbb{R}^{m_I+m_E}} \ \inf_{l \leqslant (x,y) \leqslant u} \ \ell_r(x,y,\lambda). \tag{1.8}$$

The AL algorithm is the dual algorithm that solves (QP') by solving (1.8). The phrase "dual algorithm" means that the method foccusses on the generation of the dual variables λ.

The *AL algorithm* generates a sequence of dual variables $\{\lambda_k\}_{k\in\mathbb{N}} \subset \mathbb{R}^{m_I+m_E}$ as follows.

Algorithm 1.1 (AL) Let be given $\lambda_0 \in \mathbb{R}^{m_I+m_E}$ and $r_0 > 0$. Repeat for $k \in \mathbb{N}$.

1. *Minimization of the AL*: if any, compute a solution (x_{k+1}, y_{k+1}) to

$$\min_{l \leqslant (x,y) \leqslant u} \ell_{r_k}(x, y, \lambda_k) \tag{1.9}$$

   otherwise, exit with a *direction of unboundedness* d, hence satisfying (1.4).
2. *Update the multiplier*:

$$\lambda_{k+1} = \lambda_k + r_k \begin{pmatrix} A_I x_{k+1} - y_{k+1} \\ A_E x_{k+1} - b_E \end{pmatrix}. \tag{1.10}$$

3. *Stopping criterion*: stop if the constraint norm

$$\left\| \begin{pmatrix} A_I x_{k+1} - y_{k+1} \\ A_E x_{k+1} - b_E \end{pmatrix} \right\|$$

   no longer decreases significantly.
4. *Update of the penalty parameter*: determine $r_{k+1} > 0$.

Therefore, the algorithm implemented in `Oqla` decomposes the original problem (1.1) in a sequence of subproblems (1.9), called the *AL subproblem*. The AL algorithm deserves some comments.

1. Luckily, the AL subproblems are less difficult than the original problem (1.1) since they only have bound constraints on the variables. This feature of the feasible set allows for the use of projections. Actually, the algorithm implemented in `Oqla` solves (1.9) by a combination of gradient projection (GP) steps and an active set (AS) method. A distinctive aspect of the implementation comes, however, from the special role played by the variable y in the subproblem. The Hessian of the AL with respect to y is indeed a multiple of the identity matrix, which makes easy the computation of the minimizer of the AL with respect to y.
2. The AL subproblem is always feasible, so that it has a solution if and only if it is bounded. If unbounded its directions of unboundedness are the same as those of the closest feasible problem (1.3) (whatever is λ_k and $r_k > 0$, see proposition 2.5 in [6]), so that these can be detected during the minimization of the first AL subproblem. Therefore the AL algorithm is well defined if and only if

$$\text{the closest feasible problem (1.3) has a solution.} \tag{1.11}$$

3. The AL subproblem may have several solutions (x_{k+1}, y_{k+1}) but all of them give the same value to the constraints $(A_I x_{k+1} - y_{k+1},\ A_E x_{k+1} - b_E)$, so that the new multiplier in step 2 is independent of the chosen solution (x_{k+1}, y_{k+1}).

In the next sections, we describe how the AL subproblem (1.9) is solved, alternating between gradient projection and conjugate gradient phases.

1.3.2 The gradient projection phase

The gradient projection phase consists in forcing the decrease of $\varphi_\lambda : x \in \mathbb{R}^n \mapsto \varphi_\lambda(x) := \ell_r(x, \check{y}(x), \lambda) = \inf_{y \in [l_I, u_I]} \ell_r(x, y, \lambda) \in \overline{\mathbb{R}}$, along the projected gradient path

$$p : \alpha \longmapsto p(\alpha) := P_{[l_B, u_B]}\left(x - \alpha \nabla \varphi_\lambda(x)\right)$$

`Oqla` implements two strategies. The first one consists in making only one phase of gradient projection and next to make one or more conjugate gradient phases This strategy is activated by using the options `gp_succ = one`.

The second one is inspired of the *Moré and Toraldo* method and consist in making several gradient projection phases, until one of these three criterions is satisfied:

- "The activated face does not change",
- "The decrease of the projected gradient is too small",
- "The decrease of the projected gradient is too large compared with the previous conjugate gradient phase".

It can be activated by setting the option `gp_succ = mt`.

1.3.3 The conjugate gradient phase

The goal of the minimization (CG) phase is to implement an *active set method* to minimize the augmented Lagrangian for (x, y) in the box $[l, u]$, problem (1.9). At each stage of this minimization process, some of the variables $(x, y)_i$ are fixed to one of the bounds l_i or u_i. The collection of these indices is called the *working set* and is denoted by

$$W \subset B \cup I \qquad \text{and} \qquad W := W^l \cup W^u,$$

where W^l (resp. W^u) is the set of indices i such that $(y, u)_i$ is fixed to l_i (resp. to u_i). We note

$$\begin{array}{lll} W_B^l := W^l \cap B, & W_B^u := W^u \cap B, & W_B := W \cap B = W_B^l \cup W_B^u, \\ W_I^l := W^l \cap I, & W_I^u := W^u \cap I, & W_I := W \cap I = W_I^l \cup W_I^u. \end{array} \tag{1.12}$$

The collection of the indices that are not in W is denoted by

$$V := (B \cup I) \setminus W.$$

We also introduce

$$V_B := V \cap B \qquad \text{and} \qquad V_I := V \cap I. \tag{1.13}$$

The working set W can be determined in many ways. What is important for the convergence of the algorithm that solves the AL subproblem is that the minimization phase decreases the augmented Lagrangian; then the convergence can be ensured by the gradient projection iterations that occur between the minimization phases. The determination of the working set W depends on the strategy implemented. `Oqla` implements two strategies: the *hit-and-fix* (activated by seetting the oprion `cg_as = cg_hf`) strategy and the *Moré-Toraldo* strategy (activated by seetting the oprion `cg_as = cg_mt`).

The hit-and-fix strategy

In the hit-and-fix strategy, the iterates (x, y) are maintained in the feasible set $[l, u]$. If, during the minimization phase, a bound is hit by some variables, these are fixed to the corresponding bound. The method, then proceeds with more fixed variables until a minimum is reached in the associated faces of the feasible set.

Solve the problem

$$\inf_{(x,y)_V \in [l_V, u_V]} \ell_r(x, y, \lambda). \tag{1.14}$$

without bound contraints is equal to solve

$$\boxed{\begin{aligned}&\Big(H_{V_B V_B} + r A^{\mathsf{T}}_{(W_I \cup E)V_B} A_{(W_I \cup E)V_B}\Big) x_{V_B} \\ &\qquad = A^{\mathsf{T}}_{(W_I \cup E)V_B} z_{W_I \cup E} - g_{V_B} - \Big(H_{V_B W_B} + r A^{\mathsf{T}}_{(W_I \cup E)V_B} A_{(W_I \cup E)W_B}\Big) x_{W_B},\end{aligned}} \tag{1.15}$$

For more details, see [13].

Let $(x^0, y^0) \in [l, u]$ be the iterate on entry into the minimization. The CG iterations solve (1.15) in x_{V_B}, starting from $x^0_{V_B}$, until a bound is hit in x or y.

The Moré-Toraldo strategy

The Moré-Toraldo strategy for the minimization phase in Oqla and Qpalm, called that way because it is largely inspired from the one proposed in [21], can be described in vague terms as follows. The strategy also aims at solving problem (1.14), but the CG iterations are not stopped as soon as an iterate crosses the boundaty of $[l_V, u_V]$, like in the hit-and-fix strategy. Rather, an interruption occurs when the following criterion is satisfied:

$$T_1 \quad \text{and} \quad (T_2 \quad \text{or} \quad T_3), \tag{1.16}$$

where the tests T_i can be vaguely expressed as follows

- $T_1 =$ "ℓ_r no longer decreases sufficiently" (this includes the case when a minimizer has been obtained),
- $T_2 =$ "the current iterate is outside $[l_V, u_V]$",
- $T_3 =$ "the current iterate is inside $[l_V, u_V]$ but the activated face does not look like to be the optimal one".

After satisfaction of the test (1.16) by the current iterate, a projected search determines the final iterate of the minimization phase. This operation ensures feasibility of the final iterate and a decrease of the AL. For more details about this strategy, please see [13].

1.3.4 Preconditioning of the CG iterations

We address in this section the question of the preconditioning of the linear system (1.15). Let us denote the matrix of that system by

$$\begin{aligned} M \equiv M(V_B, W_I) &= H_{V_B V_B} + r A^{\mathsf{T}}_{(W_I \cup E)V_B} A_{(W_I \cup E)V_B}, \\ &= H_{V_B V_B} + r A^{\mathsf{T}}_{EV_B} A_{EV_B} + r \sum_{i \in W_I} A^{\mathsf{T}}_{iV_B} A_{iV_B}. \end{aligned} \tag{1.17}$$

In that code, we implement two preconditioners that are quickly explain below. If you want to have nore details, please see [13].

Diagonal preconditioner

One takes the inverse of Diag(M). This diagonal preconditioner is rather easy to construct, but is not always very efficient since it does not consider an important part of the matrix M.

Cholesky preconditioner

When $M \succcurlyeq 0$, there is a lower triangular matrix L and a permutation matrix Q such that

$$QMQ^{\mathsf{T}} = LL^{\mathsf{T}}. \tag{1.18}$$

See for example Higham [17; théorème 10.9].

The system can be preconditionned by the $P = M^{-1} = Q^{-1}L^{-\mathsf{T}}L^{-1}Q^{-\mathsf{T}}$.

This preconditioner is quite efficient, but is heavy to make and update. For more detail on this preconditioner, please see [13].

2 The package

2.1 Description

The Oqla package is formed of files and directories described below.

- The directory `bin` dosen't exist originally. Once the code is compiled, it contains the execution file `main` that tests the use of `Oqla`.
- The directory `CUTEst` contains all the files that are useful to install `Oqla` in `CUTEst`[15]; see how to proceed in section 2.2.
- The directory `doc` contains all documentation files available id pdf.
- The directory `examples` gives some elementary examples.
- The directory `include` contains all the header files that are useful to compile the code.
- the directory `lib` dosen't exist originally. Once the code is compiled, it contains the library file `liboqla.a` of `Oqla`.
- The file `Makefile` is used to compile the code.
- The directory `obj` dosen't exist originally. Once the code compile, it contains all the `*.o` files of the code.
- The directory `src` contains the source code of `Oqla`.

2.2 Installation

To install `Oqla`, follow the following steps.

1. Normally, the `Oqla` package is ditributed as a tarball named

   ```
   OQLA.tgz
   ```

   Place this tarball in the directoy where you want to keep `Oqla`. Let us call it the *oqla directory*.

2. Decompress and untar it by issuing the command:

   ```
   tar -zxvf OQLA.tgz
   ```

3. To compile the code, you will need the two libraries: `Blas` [8, 9, 10, 11, 20] and `Lapack` [1]. If you already have them, skip this step and go to the next one.

   If you don't have them, you can download them on the sites

   ```
   http://www.netlib.org/blas
   http://www.netlib.org/lapack
   ```

4. You are now ready to compile `Oqla`. Go into the *oqla directory* and type

   ```
   make
   ```

   This command compiles the code and creates the file `liboqla.a` which is the library you have to link to use `Oqla`.

5. If you want to use `Oqla` in `CUTEst`, make sure that `CUTEst` is correctly installed and the environment variables are correctly set. Then go into the directory `CUTEst` and issue the command:

   ```
   make
   ```

`Oqla` is now ready to be used. As said above, these commands place the `Oqla` library in the directory `lib` and the heaser files are in the directory `include`. They can now be linked and include to a program that uses `Oqla` as a convex quadratic optimization solver.

3 Usage

We sart in section 3.1 by describing the data structures used by `Oqla`. In section 3.2, we specify the functions that are useful to run `Oqla` and describe the arguments of these functions and we give a simple example of the use of the solver.

3.1 Data structures

To run `Oqla`, you need to use the function `OQLA_Solver`. This method is associated to the class `Data`, which handle the problem data, and that we describe in section 3.1.3. The problem data is made of vectors and matrices. These two structures are described below in sections 3.1.1 (vectors) and 3.1.2 (matrices). `Oqla` can be customized by several options that are held in the structure `Options` and described in section 3.1.4. Some of these options aims to use a preconditionner. The use of a preconditionner is explain in section 3.1.5. Once the solver ran, the result diagnosis are held in the structure `Info` which is described in section 3.1.6.

3.1.1 Vectors

In `Oqla`, the `Vector` structure is used to store the vectors g, x, l_B, u_B, l_I, u_I and b_E. A `Vector` is made of an integer `mSize` that is the size of the vector and an array of double `mValues` of size `mSize` that holds the values of the vector.

There are three methods available in this class to define a Vector:

- `Vector v(n)` creates a vector of size `n` (if `n` is not provided, the default value is 0) and filled with zeros,

- `Vector v(n,val)` creates a vector of size `n` and fills it with the values in the array `val`,
- `Vector v(w)` creates a vector a vector `v` having the same size and values as `w`.

A vector `v` can be modified by using the command `v.setValue(i,val)`. In that case, the `i`th entry of the vector `v` is replaced by the value `val`.

Some other functions on the vectors can be applied. For more information, please read the full documentation.

3.1.2 Matrices

In `Oqla`, the structure `Matrix` is used to store the matrices H, A_I, and A_E.

The matrices can be stored in several formats. The two implemented inhere are the `Dense` one and the `Sparse` one. Both are devided into two subformats, one for the symmetric matrices and one for the general case. The user can also define his own Matrix format.

Dense storage formats

A $m \times n$ matrix `A`, symmetric or general, in the dense format is stored as a compact dense matrix by columns. A dense matrix is composed of three argument. The number of row (`mNrow`) and the number of column (`mNcol`) are stored as integers. The values of the entries are stored in a one-dimensional array of size $m \times n$, inwhich the component $j * n + i$ holds the value $a_{i,j}$.

The main method to create a Matrix is to use the following function:

- `DenseGenMatrix M(m,n,val)` to create a general matrix `M` of size `(m,n)` filled with the values of `val`.
- `DenseSymMatrix M(n,val)` to create a general matrix `M` of size `(n,n)` filled with the values of `val`.

Sparse storage formats

A $m \times n$ matrix `A`, symmetric or general, in the sparse format is stored in the triplet format. Only the nonzero entries are stored. The sparses matrices are composed of five arguements. The number of row and the number of column of the matrix are store as integer in the fields `mNrow` and `mNcol` respectively, as the number of nonzeros, which is stored in the field `mNnz`. The values are store into three `mNnz` dimensional array. The lth entry of the matrix a, which is at row i, column j and holds the value $a_{i,j}$ is stored in the lth componants of `mRowInd`, `mColInd` and `mValues`. If the matrix is symmetric, only the lower triangular part is stored and the number of nonzeros on the lower triangular matrix is stored in the filed `mNzlo`.

There are two main way to create a sparse matrix, one using an array containing all the values (even de zeros) and one using three arrays (one containing the indices of rows, one containing the indices of columns and one containing th values). Here is how to call the functions:

- `SparseGenMatrix A(m,n,val)` to create a matrix of size `(m,n)` filled with the values `val` (the zero entries are not stored)
- `SparseSymMatrix A(n,val)` to create a matrix of size `(n,n)` filled with the values `val` (the zero entries are not stored)

- `SparseGenMatrix A(m,n,nnz,rind,cind,val)` to create a matrix of size `(m,n)` with `nnz` nonzero entries and `A.mRowInd`, `A.mColInd` and `A.mValues` filled with `rind`,`cind` and `val`
- `SparseSymMatrix A(n,nnz,rind,cind,val)` to create a matrix of size `(n,n)` with `nnz` nonzero entries and `A.mRowInd`, `A.mColInd` and `A.mValues` filled with `rind`,`cind` and `val`

User's storage formats

Our code allow the user to implement his own matrix class. In this section, we will explain how to implement and use this feature.

To use his own matrix class, the user need to create two following files:

```
MyClass.h
MyClass.cpp
```

The file `MyClass.h` must have the following shape:

```
#include "Vector.h"
#include "Option.h"
#include "Info.h"
#include "Matrix.h"

class MyClass : public Matrix
{
...
}
```

All the methods that are in the file `Matrix.h` need to be defined (in the file `MyClass.h`) and implemented (in the file `MyClass.cpp`).

Once the files are written, place `MyClass.h` into the directory `include` and the `MyClass.cpp` into the directory `src`. You need to recompile the code to take into account the changes. To do that see section 2.2.

After compilation, you can create a matrix of class `MyClass` and run the solver using it.

3.1.3 Problem data

The structure `Data` must be used to store the data of the problem. The componants of this class, describing the problem, are these ones:

- `int n`: Number of variables.
- `int mi`: Number of inequality constraints.
- `int me`: Number of equality contraints.
- `Vector g`: `Vector` of size `n` used to store the vector $g \in \mathbb{R}^n$ giving the linear part of the objective of the problem.
- `Matrix *H`: `Matrix` of size `n`×`n` used to store the Hessian H of the objective of the problem.
- `Vector lb`: `Vector` used to store the vector l_B giving the lower bounds on the variables x.
- `Vector ub`: `Vector` used to store the vector u_B giving the upper bounds on the variables x.

- `Matrix *Ai`: `Matrix` of size `mi`×`n` used to store the Jacobian A_I of the inequality constraints of the problem.
- `Vector li`: `Vector` used to store the vector l_I giving the lower bounds on $A_I x$
- `Vector ui`: `Vector` used to store the vector u_I giving the upper bounds on $A_I x$
- `Matrix *Ae`: `Matrix` of size `me`×`n` used to store the Jacobian A_E of the equality constraints of the problem.
- `Vector e`: `Vector` used to store the vector $e \in \mathbb{R}^{m_E}$ giving the RHS of the equality constraints.

3.1.4 Solver options

The structure `Option` is used to describe the options of the Oqla solver, i.e. the parameters that can be used to tune the behavior of the solver. The components of this structure are describe below, ordering in alphabetic order and valid and default values are indicated.

- `accu`: Number specifying the requires accuracy of the computed residual at the final iteration, compared with the exact residual.
  Valid value: $[0, 1]$
  Default value: 10^{-1}
- `alit`: Maximal number of AL iterations.
  Valid value: > 0
  Default value: 100
- `cg_as`: Option that indicates which strategy has to be used in the conjugate gradient phases (see 1.3.3).
  Valid values: `cg_hf`, `cg_mt`
  Default value: `cg_hf`
- `cgit`: Maximal number of CG iterations.
  Valid value: > 0
  Default value: `INT_MAX`
- `cgprec`: Flag that indicates which preconditioner is used. The following values are meaningful:
  - `none`: no preconditioner is used;
  - `diag`: the diagonal preconditioner is used;
  - `cholesky`: the Cholesky preconditioner is used.

  Valid value: `none`, `diag`, `cholesky`
  Default value: `none`
- `cput`: Scalar that is used to determine the maximum permitted CPU time. A negative value indicates no limit.
  Valid value: $[-\infty; \infty]$
  Default value: -1.0
- `dcr`: Scalar that indicates the desired convergence rate.
  Valid value: $]0, 1[$
  Default value: 10^{-1}
- `dcrf`: This is the minimum value for the decrease of the constraints. If $\log(cn_i/cn_{i-1}) <$

lrhomin, the solver stops. That indicates that no better solution can be found.
Valid value: $[0, 1[$
Default value: 10^{-2}

- `dgp`: Scalar that indicates the non-pregress for the GP phase (only used if options.MT = true).
  Valid value: $]0, 1]$
  Default value: 0.25
- `dxmin`: Resolution in x. Two point x and x' such that $\|x - x'\|_\infty) <= dxmin$ are consideresd identical in a linesearch.
  Valid value: > 0
  Default value: 10^{-15}
- `dymin`: Resolution in y. Two point y and y' such that $\|y - y'\|_\infty) <= dymin$ are consideresd identical in a linesearch.
  Valid value: > 0
  Default value: 10^{-15}
- `feas`: Required accuracy on the constraint feasibility.
  Valid value: > 0
  Default value: 10^{-8}
- `feass`: Required accuracy on the shifted constraint feasibility.
  Valid value: > 0
  Default value: 10^{-8}
- `fout`: Output channel. this is the name of the output file. to print the ouputs on the screen use "*screen*".
  Valid value: any string
  Default value: `"screen"`
- `gp_succ`: Option that indicates which strategy has to be used in the gradient projection phases (see 1.3.2).
  Valid value: `one`, `mt`
  Default value: `one`
- `gtol`: Required accuracy on the gradient of the objective.
  Valid value: > 0
  Default value: 10^{-8}
- `inf`: Value beyond which a number is considered infinite.
  Valid value: > 0
  Default value: `INFINITY`
- `MT`: Boolean indicating if the Moré-Toraldo method is used. In that case, options `gp_succ` and `cg_as` are respectively set to `'mt'` and `'cg_mt'`.
  Valid value: `true`, `false`
  Default value: `false`
- `npcg`: Used only if `options.MT = true`. This is the value that indicate a non progress for conjugate gradient.
  Valid value: $]0, 1]$

Default value: 0.1

- `phase`: Maximum number of phases.
  Valid value: > 0
  Default value: `INT_MAX`
- `ps`: Used only if `options.MT = true`. This is the parameter for the prjected search.
  Valid value: $[0, 1]$
  Default value: 0.1
- `rctl`: Type of control for the augmented parameter r. The following values are meaningful: - fixed: augmentation parameter r is maintained fixed during all the run; - constraint: augmentation parameter r is updated to have the desired convergence rate options.dcr of the constraits values; - shifted_constraint: augmentation parameter r is updated to have the desired convergence rate option.dcr of the constrait values shifted by the (estimated) smallest feasible shift.
  Valid value: `fixed`, `constraint`, `shifted_constraint`
  Default value: `shifted_constraint`
- `rfct`: Only used if `options.cgprec = cholesky`. This is the modification factor for the Cholesky factorization.
  Valid value: $\geqslant 1$
  Default value: 1
- `rlag`: Initial value of the augmentation parameter.
  Valid value: > 0
  Default value: 1
- `ubtol`: Small nonegative number to detect unboundedness.
  Valid value: > 0
  Default value: $n\varepsilon$
- `verb`: Verbosity level of the solver. The following values are meaningful: $= 0$: Nothing is written; $= 2$: In addition to the initial and final output, the solver writes one line at each AL iteration; $= 3$: Details from gradient projection and conjugate gradient phases are given; $\geqslant 4$: More details are given (for example active bounds).
  Valid value: $\geqslant 0$
  Default value: 0

3.1.5 Preconditioners

As said in section 3.1.4, the field `cgprec` can be set to `'none'`, `'diag'` or `'cholesky'`. The preconditioner is used to decrease the condition number of a matrix and so the computation time decrease too.

In the code, it exists a class named `precond`. This class has, until now, two derived classes, one for each preconditioner. These derived classes have at least two arguments and four methods. The arguments are:

- `mRank`: this is an integer that represent the rank of the preconditioner
- `mPrec`: This is an objectthat can be of various type and that represents the preconditioned matrix. In the case `'diag'`, this is a vector while in the case `'cholesky'` this is a matrix.

The methods of the derived classes are:

- `getRank`: this method provides the rank of the preconditioner.
- `prmk`: this is the preconditionner maker. In particular, it provides the preconditionner matrix.
- `prec`: it computes $P * v$ where P is the preconditionned matrix and v is a vector. This vector give the descent direction in the conjugate gradient phase. If a zero curvature direction is detected, it is this direction that is returned.
- `prup`: it update the preconditioner based on the activated bounds. It is faster than to make the preconitioner.

3.1.6 Solver diagnostsis

The structure `Info` is used to define the variable getting information in the problem on return from the solver. The components of this structure are describe below, ordering in alphabetic order.

- `accu`: Accuracy of the CG linear solver.
- `alit`: Total number of AL iterations.
- `ccgp`: Number of consecutive CG phases.
- `CG_dec`: The decreasing between two consecutive CG phases.
- `cgit`: Total number of conjugate-gradient iterations.
- `cgph`: Total number of CG phases.
- `cput`: CPU time spent in the solver.
- `elapsedTime`: The elapsed time in the program.
- `f`: Value of the objective at the final point (meaningful if `info.flag=0`).
- `feas`: Inf-norm of the constraints.
- `feass`: The closest feasibility.
- `flag`: Specifies the status of the QP solver on return. The following values are possible:

  0 = Solution found
  1 = success shifted
  2 = unbounded problem
  3 = the feasibility tolerance options.feas cannot be reached
  4 = gtol unreachable (too stringent optimality tolerance)
  5 = erroneous argument
  6 = max AL iterations reached
  7 = max phases reached
  8 = max number of GP phases reached
  9 = max number of CG phases reached
  10 = max CG iterations reached
  11 = time limit reached
  12 = non convex problem
  13 = fail on accuracy (r too many times reduced)
  14 = not computed

  The flag `info` can be set to these other values when the solver is runnig, but not for the final diagnosis.

  15 = relative minimum found (used in CG phases)

16 = gp successful (used in GP phases)
17 = stepsize failure (stop on dxmin during linesearch)
18 = bound hit (used in CG phases)
19 = the GP does not change the face (used if `options.gp_succ = mt`)
20 = the GP does not progress enough (used if `options.gp_succ = mt`)
21 = stop GP (used if `options.gp_succ = mt`)
22 = bound hit in CG (used if `options.cg_as = cg_mt`)
23 = not the optimal face (used if `options.cg_as = cg_mt`)
24 = a solution has been found in the CG phase (used if `options.cg_as = cg_mt`)
25 = next do a CG phase (used if `options.cg_as = cg_mt`)

- `glan`: The norm of the gradient of the Lagrangian at the final primal-dual pair
- `gopt`: Inf-norm of the gradient of the objective
- `gpit`: Total number of GP phases
- `gpph`: Number of GP phases in the last AL subproblem
- `lips`: Estimated value of the lipschitz constant
- `mnph`: Number of CG phases in the last AL subproblem
- `nb`: Number of variables subject to a bound
- `pgln`: Inf-norm of the projected gradient of the Lagrangian (inf if x is infeasible)
- `rlag`: Final augmentation parameter
- `rq`: Rayleigh quotient d'*H*d/(d'*d), where d is a direction of unboundedness (meaningful if `info.flag=5`)
- `rqmax`: Maximum Rayleigh quotient
- `rqmin`: Minimum Rayleigh quotient
- `s`: The shifted vector
- `tbegin`: The beginning time of the computation
- `ubdd`: The vector giving a direction in x along which the QP goes to go to -Inf (meaningful if `info.flag=5`)
- `wb`: The number of active x-bounds
- `wi`: The number of active inequality bounds

3.2 Running the solver

To run the solver you need to run the function `OQLA_Solver` which belongs to the `Data` class. Here is the définition statement.

```
void OQLA_Solver(Vector &x0, Vector &lm0, Option &options,
                 Info &info);
```

The arguments of this method are as follow:

- `x0`(IO): This is an object of the class `Vector` (see 3.1.1). It is a vector of size n of variables x to optimize.
  - On entry, it is the initial guess of the solution to the quadratic problem to solve. It is taken as starting point by `Oqla`. It needs not to be feasible.
  - On return, when `info.flag = 0` or `info.flag = 1`, it is the feasible point (or the closest feasible point) that minimize the objective function.
- `lm0` (IO): This is an object of the classe `Vector`. It is a vector of size $n + m_E + m_I$ and is the dual variable associated with the constraint of the quadratic problem. The

first n components are associated with the bounds on x, the next m_I are associated with the inequality contraints and then the lasr m_E are associated with the equality constraints.

  - On entry, it is the initial guess of the dual solution. to the problem.
  - On return, if `info.flag = 0` or `info.flag = 1`, then it is the of the vector optimal dual variables founded by `Oqla`.

- `options`(I): this is a variable of type `Option`, whose components are described in section 3.1.4. It is used to tune the behavior of the solver.
- `info`(O): this is a variable of type `Info`, whose components are described in section 3.1.6. It is used to get the information on the problem resolution on return from the solver.

This function `OQLA_Solve` is a method of the class `Data`. To run it, you need to apply it to an object of the class `Data`. Let us note `data` the object of the class `Data` that holds the problem data we want to solve, `options` the object of the class `Option`, `info` the object of th class `Info` and `x` and `lm` two `Vectors`. Then to run the solver, run the command:

```
data.OQLA_Solver(x, lm, options, info);
```

3.3 Example

The best way to show how to run Oqla is by providing an example. Suppose we wish to minimize the convex quadratic function $2x_1^2+2x_2^2+\frac{3}{2}x_3^2+2x_1x_3-3x_2$ subject to the linear inequality contraint $2x_2+x_3 \leqslant 1$, the linear equality contraint $x_1+x_2=2$ and the bound constraints $x_2 \leqslant 0$ and $-1 \leqslant x_3 \leqslant 1$. The problem can be written in the form (1.1) with $n=3$, $m_I=1$, $m_E=1$ and

$$g = \begin{pmatrix} 0 \\ -3 \\ 0 \end{pmatrix}, \quad H = \begin{pmatrix} 4 & 0 & 2 \\ 0 & 4 & 0 \\ 2 & 0 & 3 \end{pmatrix},$$
$$l_B = \begin{pmatrix} -\infty \\ -\infty \\ -1 \end{pmatrix}, \quad u_B = \begin{pmatrix} +\infty \\ 0 \\ 1 \end{pmatrix},$$
$$A_I = \begin{pmatrix} 0 & 2 & 1 \end{pmatrix}, \quad l_I = -\infty, \quad u_I = 1,$$
$$A_E = \begin{pmatrix} 1 & 1 & 0 \end{pmatrix}, \quad \text{and} \quad b_E = 2.$$

To solve this problem, we may use the following code.

```
#include "cblas.h"
#include <cstring>
#include <string>
#include <cmath>
#include <sys/time.h>
#include <limits>

#include "Data.h"
#include "Info.h"
#include "Option.h"
#include "Vector.h"
```

```
#include "Matrix.h"

int main(){

// Defining the problem dimensions

    int n  = 3;          // Number of variables
    int mi = 1;          // Number of inequality contraints
    int me = 1;          // Number of equality contraints

// Defining the arrays containing the data

    double gd[3]  = {0, -3, 0};
    double Hd[9]  = {4, 0, 2, 0, 4, 0, 2, 0, 3};
    double lbd[3] = {-INFINITY,-INFINITY, -1};
    double ubd[3] = {INFINITY,0, 1};
    double Aid[3] = {0,2,1};
    double lid[1] = {-INFINITY};
    double uid[1] = {1};
    double Aed[3] = {1, 1, 0};
    double bed[1] = {2};

// Construction of the Data Vectors and Matrices

    Vector g(n,gd);
    Vector lb(n,lbd), ub(n,ubd);
    Vector li(mi,lid), ui(mi,uid);
    Vector be(me,ed);
    Matrix *H  = new DenseSymMatrix(n,Hd);
    Matrix *Ae = new DenseGenMatrix(1,me,Aed);
    Matrix *Ai = new DenseGenMatrix(1,3,Aid);

// Construction of the Data object

    Data prob(n,mi,me,g,H,lb,ub,Ai,li,ui,Ae,be);

// Defining the Option structure and possible changes

    Option options;
    options.verb = 4;   // Verbosity level is now 4

// Defining the Info structure

    Info info;

// Construction of the primal and dual vectors and
// possible initializations

    Vector x, lm;

// Solving the problem

    prob.OQLA_Solver(x,lm,options,info);

// Freeing allocated memory

    delete H;
    delete Ai;
    delete Ae;
```

```
}
```

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen (1999). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition. 9

[2] K.J. Arrow, F.J. Gould, S.M. Howe (1973). A generalized saddle point result for constrained optimization. *Mathematical Programming*, 5, 225–234. [doi]. 4

[3] A. Auslender, M. Teboulle (2003). *Asymptotic Cones and Functions in Optimization and Variational Inequalitites*. Springer Monographs in Mathematics. Springer, New York. 3

[4] 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]. 3

[5] J.D. Buys (1972). *Dual algorithms for constrained optimization*. PhD Thesis, Rijksuniversiteit te Leiden, Leiden, The Netherlands. 4

[6] A. Chiche, J.Ch. Gilbert (2014). How the augmented Lagrangian algorithm can deal with an infeasible convex quadratic optimization problem. *Journal of Convex Analysis* (to appear). [pdf]. 3, 4, 5

[7] F. Delbos, J.Ch. Gilbert (2005). Global linear convergence of an augmented Lagrangian algorithm for solving convex quadratic optimization problems. *Journal of Convex Analysis*, 12, 45–69. [preprint] [editor]. 4

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, I. S. Duff (1990, March). A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1), 1–17. [doi]. 9

[9] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson (1988, March). Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw.*, 14(1), 18–32. [doi]. 9

[10] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson (1988, March). An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1), 1–17. [doi]. 9

[11] J. J. Dongarra, Jermey Du Cruz, Sven Hammerling, I. S. Duff (1990, March). Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw.*, 16(1), 18–28. [doi]. 9

[12] M. Frank, P. Wolfe (1956). An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3, 95–110. [doi]. 2

[13] J.Ch. Gilbert, E. Joannopoulos. Inside Oqla and Qpalm. Research report, INRIA, BP 105, 78153 Le Chesnay, France. (to appear). 7, 8

[14] J.Ch. Gilbert, É. Joannopoulos (2014). OQLA/QPALM - Convex quadratic optimization solvers using the augmented Lagrangian approach, with an appropriate behavior on infeasible or unbounded problems. Technical report, INRIA, BP 105, 78153 Le Chesnay, France. (to appear). 3

[15] N. I. M. Gould, D. Orban, P. L. Toint (2013). CUTEst : a Cconstrained and Unconstrained Testing Environment with safe threads. Technical report, Rutherford Appleton Laboratory Chilton, Oxfordshire OX11 0QX, England.
http://ccpforge.cse.rl.ac.uk/gf/project/cutest/wiki/. 8

[16] M.R. Hestenes (1969). Multiplier and gradient methods. *Journal of Optimization Theory and Applications*, 4, 303–320. [doi]. 4

[17] N.J. Higham (2002). *Accuracy and Stability of Numerical Algorithms* (second edition). SIAM Publication, Philadelphia. 8

[18] J.-B. Hiriart-Urruty, C. Lemaréchal (1996). *Convex Analysis and Minimization Algorithms* (second edition). Grundlehren der mathematischen Wissenschaften 305-306. Springer-Verlag. 3

[19] A.F. Izmailov, M.V. Solodov (2014). *Newton-Type Methods for Optimization and Variational Problems*. Springer Series in Operations Research and Financial Engineering. Springer. 3

[20] C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh (1979, September). Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3), 308–323. [doi]. 9

[21] J.J. Moré, G. Toraldo (1991). On the solution of large quadratic programming problems with bound constraints. *SIAM Journal on Optimization*, 1, 93–113. [doi]. 7

[22] M.J.D. Powell (1969). A method for nonlinear constraints in minimization problems. In R. Fletcher (editor), *Optimization*, pages 283–298. Academic Press, London. 4

[23] R.T. Rockafellar (1970). *Convex Analysis*. Princeton Mathematics Ser. 28. Princeton University Press, Princeton, New Jersey. 3

[24] R.T. Rockafellar (1971). New applications of duality in convex programming. In *Proceedings of the 4th Conference of Probability, Brasov, Romania*, pages 73–81. 4

[25] R.T. Rockafellar (1973). The multiplier method of Hestenes and Powell applied to convex programming. *Journal of Optimization Theory and Applications*, 12, 555–562. [doi]. 4

[26] R.T. Rockafellar (1974). Augmented Lagrange multiplier functions and duality in nonconvex programming. *SIAM Journal on Control*, 12, 268–285. 4

Index