SQPlab – A Matlab software for solving nonlinear optimization problems and optimal control problems

Version 0.4.4 (February 2009)

J. Charles Gilbert[†]

The name of the code, SQPlab, stands for Sequential Quadratic Programming (SQP) laboratory. It is written in Matlab. This piece of software is used by the author as a kind of laboratory for trying techniques related to the SQP algorithm, but it has been designed so that it can be useful to many. It is also aimed at accompanying the *hanging chain project* developped along part III of the book [1], during which it is shown how to implement step by step many aspects of the algorithm. The code is still in an embryonic stage: one of its main defects is that

[†]INRIA-Rocquencourt, BP 105, F-78153 Le Chesnay Cedex, France; e-mail: `Jean-Charles.Gilbert@inria.fr`.

SQPlab is presently unable to deal with incompatible linearized constraints; if this situation is encountered, the solver simply deaclares failure.

1 The optimization problem to solve

1.1 The problem definition

SQPlab can solve a general nonlinear optimization problem of the form

$$(P) \quad \begin{cases} \min_{x\in\mathbb{R}^n} \ f(x) \\ l \leqslant (x, c_I(x)) \leqslant u \\ c_E(x) = 0 \\ c_S(x) = 0, \end{cases} \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $c_I : \mathbb{R}^n \to \mathbb{R}^{m_I}$, and $c_E : \mathbb{R}^n \to \mathbb{R}^{m_E}$ are nonlinear smooth functions, possibly nonconvex. Smoothness means that at least first order differentiability is required. The notation $l \leqslant (x, c_I(x)) \leqslant u$ expresses in compact form *bound constraints* on x and on $c_I(x)$. The bounds l and $u \in \bar{\mathbb{R}}^{n+m_I}$ must verify $l < u$ and may have components with infinite values (they are not considered in that case). On the other hand, problem (P) can have $m_I \geqslant 0$ nonlinear *inequality constraints* and $m_E \geqslant 0$ *equality constraints*. The function $c_S : \mathbb{R}^n \to \mathbb{R}^{m_S}$ imposes additional equality constraints that are treated by SQPlab in a way that is appropriate to *optimal control problems* (see section 1.2); these $m_S \geqslant 0$ constraints are named *state constraints* below.

A solution to problem (P) is a vector x_*, made of n components, that is *feasible* (i.e., that satisfies the constraints of problem (P)) and that gives to f a value not greater than the one given by any other feasible point (or vector).

To be concise, we note $c_B(x) \equiv x$ and $c : \mathbb{R}^n \to \mathbb{R}^m$ the function defined by $c(x) := (c_B(x), c_I(x), c_E(x), c_S(x))$; hence $m := n + m_I + m_E + m_S$. Occasionally, we will denote by n_B the number of variables x_i that are subject to a finite lower and/or upper bound. For a vector $v \in \mathbb{R}^m$, we define the vector $v^\#$ by

$$(v^\#)_i = \begin{cases} \max(0, l_i - v_i, v_i - u_i) & \text{if } i \in B \cup I \\ v_i & \text{if } i \in E \cup S. \end{cases} \tag{2}$$

Then all the constraints of (P) can be written $c(x)^\# = 0$. This does not make the problem easier, however, since $x \mapsto c(x)^\#$ is usually non differentiable. It is just a way of making the notation more concise.

The *Lagrangian* of problem (1) is the function $\ell : \mathbb{R}^n \times \mathbb{R}^m$ defined at (x, λ) by

$$\ell(x, \lambda) = f(x) + \lambda^\top c(x). \tag{3}$$

This one is useful to write the KKT *optimality conditions* of problem (P). Note that, for $i \in B \cup I$, λ_i is actually the difference between the multiplier associated with the upper bound constraint $c_i(x) \leqslant u_i$ and the one with the lower bound constraint $l_i \leqslant c_i(x)$. If x is a *solution* to (P) and if the constraints are qualified

at x, there exists a vector $\lambda \in \mathbb{R}^m$ such that:

$$\left\{ \begin{array}{ll} \text{(a)} & \nabla_x \ell(x,\lambda) = 0 \\ \text{(b)} & l \leqslant (x, c_I(x)) \leqslant u, \quad c_E(x) = 0, \quad c_S(x) = 0 \\ \text{(c)} & \forall i \in B \cup I: \quad \lambda_i^-(l_i - c_i(x)) = \lambda_i^+(c_i(x) - u_i) = 0, \end{array} \right. \tag{4}$$

where $t^+ := \max(t, 0)$ and $t^- := \max(-t, 0)$. In (c), infinite bounds are replaced by large numbers of the same sign, so that, for example, λ_i must be nonnegative when $l_i = -\infty$ and u_i is finite. The first condition refers to the *proper optimality*, the second one to the *feasibility*, and the third one is known as the *complementarity conditions*. The components of the vector λ in this equation are called the *optimal KKT multipliers* or the *dual solutions* or the *marginal costs*.

1.2 State constraints

The constraint $c_S(x) = 0$ can be used to express *state constraints* in an optimal control setting. These constraints are assumed to be without singularities, in the sense that the Jacobian matrix

$$A_S(x) = c_S'(x)$$

is assumed *uniformly surjective*, which means that

$$\exists \gamma_S > 0, \ \forall x \in \mathcal{X}_S, \ \forall v \in \mathbb{R}^{m_S}: \quad \|A_S(x)^\top v\| \geqslant \gamma_S \|v\|. \tag{5}$$

The x's range on a large closed set $\mathcal{X}_S$, which may not be the full space $\mathbb{R}^n$, but should include the solutions and the iterates generated by the algorithm.

When $A_S(x)$ is surjective, it has a *right inverse* and it is assumed that this one is the value at x of a smooth map

$$A_S^- : \mathcal{X}_S \to \mathbb{R}^{n \times m_S} : x \mapsto A_S^-(x).$$

Hence

$$\forall x \in \mathcal{X}_S\text{: } A_S^-(x) \text{ is injective and } A_S(x)A_S^-(x) = I_{m_S}. \tag{6}$$

On the other hand, the null space $\mathcal{N}(A_S(x))$, which is also the space tangent to the manifold $\{x' \in \mathbb{R}^n : c_S(x') = c_S(x)\}$ at x, has a basis formed of $n - m_S$ vectors. We assume that these vectors are given by a smooth map

$$Z_S^- : \mathcal{X}_S \to \mathbb{R}^{n \times (n - m_S)} : x \mapsto Z_S^-(x).$$

More precisely, for all $x \in \mathcal{X}_S$, the columns of $Z_S^-(x)$ form a basis of $\mathcal{N}(A_S(x))$ or equivalently:

$$\forall x \in \mathcal{X}_S\text{: } Z_S^-(x) \text{ is injective and } A_S(x)Z_S^-(x) = 0. \tag{7}$$

If a state constraint exists ($m_S \neq 0$), SQPlab will ask the simulator to compute the products

$$A_S^-(x)v \quad \text{and} \quad Z_S^-(x)w$$

for various $x \in \mathcal{X}_S$, $v \in \mathbb{R}^{m_S}$, and $w \in \mathbb{R}^{n - m_S}$.

In optimal control problems, these functions A_S^- and Z_S^- can be deduced from a partition of the variables $x = (y, u)$ in *state variables* $y \in \mathbb{R}^{m_S}$ and *control variables* $u \in \mathbb{R}^{n-m_S}$. Consider the corresponding partition of the Jacobian $A_S(x)$:

$$A_S(x) = \begin{pmatrix} B(x) & N(x) \end{pmatrix},$$

where the square matrix $B(x)$ is supposed to be nonsingular on for $x \in \mathcal{X}_S$ with $\{B(x)\}_{x\in\mathcal{X}_S}$ and $\{B(x)^{-1}\}_{x\in\mathcal{X}_S}$ bounded. Then one can take

$$A_S^-(x) = \begin{pmatrix} B(x)^{-1} \\ 0 \end{pmatrix} \quad \text{and} \quad Z_S^-(x) = \begin{pmatrix} -B(x)^{-1}N(x) \\ I_{m_S} \end{pmatrix}.$$

2 Description of the method

One iteration of the SQP algorithm (see part III of [1] for an introduction) is made of a sequence of stages. We describe them in sequence in this section. Only the elements that are useful for understanding the behavior of SQPLAB are given.

2.1 Osculating quadratic problem

The SQP algorithm decomposes problem (P) in a sequence of quadratic subproblems (quadratic objective and linear constraints). Such a subproblem is called an *osculating quadratic problem* (OQP and QP). At the current iterate $(x, \lambda) \in \mathbb{R}^n \times \mathbb{R}^m$, it reads (we drop the dependence of the functions in (x, λ)):

$$\begin{cases} \min_{d\in\mathbb{R}^n} \; g^\top d + \frac{1}{2} d^\top M d \\ \tilde{l} \leqslant (d, A_I d) \leqslant \tilde{u} \\ c_{E\cup S} + A_{E\cup S} d = 0, \end{cases} \tag{8}$$

where g is the *gradient* $\nabla f(x)$, M is either the *Hessian of the Lagrangian* $L := \nabla^2_{xx}\ell(x, \lambda)$ or an approximation to it, $A_I := c'_I(x)$, $A_{E\cup S} := c'_{E\cup S}(x)$, $\tilde{l} = l - c_{B\cup I}(x)$, and $\tilde{u} := u - c_{B\cup I}(x)$. It is classical to impose the positive semi-definiteness of M (even though L does not have that property), in order to have a convex OQP, which, otherwise, would be NP-hard. See section 2.2 for more details on the computation of M.

The osculating QP is solved in a way that depends on the type of globalization technique used to force convergence from remote starting points (see section 2.3). It can be either completely solved by using a general purpose QP solver (section 2.1.1) or approximately solved within a trust region (section 2.1.2).

2.1.1 Solving the OQP completely

When $S = \varnothing$, SQPLAB solves (8) thanks to the QP solver `quadprog` from the optimization toolbox of Matlab.

When $S \neq \varnothing$, SQPLAB eliminates the linearized state constraints from (8) as follows. Any solution to problem (8) verifies $c_S + A_S d = 0$, so that, with the operators $A_S^- \equiv A_S^-(x)$ and $Z_S^- \equiv Z_S^-(x)$ defined in section 1.2, it can be written

$$d = r + t, \tag{9}$$

where $r \in \mathcal{R}(A_S^-)$ is the *restoration step* and $t \in \mathcal{R}(Z_S^-)$ is the *tangent step*. The step t is indeed tangent to the manifold $c_S^{-1}(c_S(x))$, which is "parallel" to the state constraint manifold $c_S^{-1}(0)$. Clearly, there holds

$$r := -A_S^- c_S \in \mathbb{R}^n, \tag{10}$$

while $t := Z_S^- h$ with $h \in \mathbb{R}^{n-m_S}$ is a solution to the *tangent quadratic problem*:

$$\begin{cases} \min_{h \in \mathbb{R}^{n-m_S}} (g + Mr)^\top Z_S^- h + \frac{1}{2} h^\top Z_S^{-\top} M Z_S^- h \\ \tilde{l}' \leqslant (Z_S^- h, A_I Z_S^- h) \leqslant \tilde{u}' \\ c_E + A_E r + A_E Z_S^- h = 0. \end{cases} \tag{11}$$

We have denoted by $\bar{g} := Z_S^{-\top} g$ the *reduced gradient*, $\tilde{l}' = l - c_{B \cup I}(x) - A_{B \cup I} r$ and $\tilde{u}' = u - c_{B \cup I}(x) - A_{B \cup I} r$. The $(n - m_S) \times (n - m_S)$ matrix $\bar{M} := Z_S^{-\top} M Z_S^-$ is called the *reduced matrix*. The present version of the software does not allow you to solve problems with $S \neq \varnothing$ and inequality constraints.

2.1.2 Solving the OQP within a trust region

In the present version of the solver, the technique presented in this section can be used when there are only equality constraints.

A trust region is used to express that the QP is only trusted on a ball of radius Δ, the *trust radius*. In this approach the OQP (8) is replaced by the following restricted osculating QP:

$$\begin{cases} \min_{d \in \mathbb{R}^n} \left(q(x) := g^\top d + \frac{1}{2} d^\top M d \right) \\ c_E + A_E d = 0 \\ \|d\|_2 \leqslant \Delta. \end{cases} \tag{12}$$

The trust radius is determined by the algorithm and evolves from one iteration to the other to ensure global convergence. The difficulty with problem (12) is that it likely to be infeasible when Δ is small, because the Euclidean ball $B_2(0, \Delta)$ does not intersect the linearized constraint affine space $\{d \in \mathbb{R}^n : c_E + A_E d = 0\}$. In the approach proposed by Byrd and Omojokun [2, 4], which we follow, the computed step approaches a solution to (8) within the trust region as follows.

A step aiming at restoring the constraint is first computed by solving the least-squares problem

$$\begin{cases} \min_{r \in \mathbb{R}^n} \ \frac{1}{2} \|c_E + A_e r\|_2 \\ \|r\|_2 \leqslant \xi \Delta. \end{cases} \tag{13}$$

Since ξ is taken in $(0, 1)$, there is still some place to make a minimization step within the trust region. This is realized by relaxing (12) into

$$\begin{cases} \min_{d \in \mathbb{R}^n} \left(q(x) := g^\top d + \frac{1}{2} d^\top M d \right) \\ c_E + A_E d = c_E + A_E r \\ \|d\|_2 \leqslant \Delta. \end{cases} \tag{14}$$

...

2.2 Hessians and their approximation

2.2.1 Newton methods

When second derivatives are computed, the SQP algorithm is a *Newton-like method* applied to the optimality conditions. In particular, provided the starting point is close enough to a regular stationnary point (x_*, λ_*) (local convergence) and some mild assumptions are satisfied, the algorithm generates a sequence of primal-dual iterates (x_k, λ_k) that converge quadratically to (x_*, λ_*). The Newton algorithm is selected by setting (see section 3.1)

```
options.algo_method = 'Newton';
```

The implementation of the Newton method has two variants, which depend on the value of m_S.

When $m_S = 0$ (standard problems), this Newton-like method requires to take for symmetric $n \times n$ matrix M in the osculating quadratic problem (8), the *Hessian of the Lagrangian*

$$L \equiv L(x, \lambda) \equiv \nabla^2_{xx} \ell(x, \lambda), \tag{15}$$

which is the $n \times n$ symmetric matrix of the second order derivatives of the Lagrangian ℓ with respect to x. Its (i, j) element is therefore given by

$$L_{ij} = \frac{\partial^2 \ell}{\partial x_i \partial x_j}(x, \lambda). \tag{16}$$

When $m_S \neq 0$ (optimal control problems), the osculating QP becomes (11), which requires the computation of the *reduced Hessian of the Lagrangian*

$$\bar{L} := Z_S^{-\top} L Z_S^- \equiv Z_S^-(x) L(x, \lambda) Z_S^-(x)^\top \tag{17}$$

as well as the matrix-vector product

$$Lr. \tag{18}$$

2.2.2 Quasi-Newton methods

Quasi-Newton algorithms in SQPlab are selected by the setting (see section 3.1)

```
options.algo_method = 'quasi-Newton';
```

They generate approximations M_k of Hessians by updating them with the *BFGS formula* (19) below (see [1] for example). Given two well chosen vectors s_k and y_k belonging to the same space, the updated matrix M_{k+1} is given by

$$M_{k+1} = M_k - \frac{M_k s_k s_k^\top M_k}{s_k^\top M_k s_k} + \frac{y_k y_k^\top}{y_k^\top s_k}. \tag{19}$$

Since $y_k = M_{k+1} s_k$ and M_k is a Hessian approximation, it makes sense to choose for y_k the change in some gradient from x_k to $x_{k+1} = x_k + s_k$. Obviously, the

BFGS formula preserves symmetry. It also preserves positive definiteness when the *monotonicity condition*

$$y_k^\top s_k > 0 \tag{20}$$

is satisfied [3].

In some cases (essentially unconstrained problems or problems with only state constraints), the monotonicity condition can nicely be realized by (piecewise) linesearch. When the structure of the problem does not allow the linesearch to guarantee the monotonicity condition, this one is ensured by the following heuristics, known as the *Powell correction* [5] (even if there is no linesearch). If $y_k^\top s_k$ is sufficiently positive in the sense that $y_k^\top s_k \geqslant \kappa\, s_k^\top M_k s_k$, where the constant $\kappa \simeq 0.2$, the vectors y_k and s_k are used in the BFGS formula. Otherwise, y_k is replaced by

$$y_k^{\mathrm{P}} := \theta_k y_k + (1-\theta_k) M_k s_k,$$

where θ_k is the greatest number in $[0,1]$ such $(y_k^{\mathrm{P}})^\top s_k \geqslant \kappa\, s_k^\top M_k s_k$. A similar technique can be used when $W_k := M_k^{-1}$ is updated by the inverse BFGS formula to approximate some inverse Hessian. In that case, it is more appropriate to modify s_k into

$$s_k^{\mathrm{P}} := \theta'_k s_k + (1-\theta'_k) W_k y_k,$$

where θ'_k is the greatest number in $[0,1]$ such $(y_k^{\mathrm{P}})^\top s_k \geqslant \kappa\, y_k^\top W_k y_k$. This heuristics often provides good results, but can also yield ill-conditioned matrices M_k or W_k.

When $m_S = 0$ (standard problems), the matrices M_k try to approximate at best the full Hessian of the Lagrangian $L(x_k, \lambda_k)$. If there is no constraint (or active affine constraints), the monotonicity condition (20) is ensured by the *Wolfe linesearch*, which provides appropriate vectors $s_k = x_k + \alpha_k d_k$ and $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ satisfying the monotonicity condition (20) (see section 2.3). In the presence of constraints, one should take for y_k the change in the gradient of the Lagrangian

$$y_k^\ell = \nabla_x \ell(x_{k+1}, \lambda_{k+1}) - \nabla_x \ell(x_k, \lambda_{k+1}).$$

However, since the Lagrangian may never have a positive curvature along d_k, even close to the solution, the monotonicity condition $(y_k^\ell)^\top s_k > 0$ cannot be ensured by linesearch along d_k. Therefore, the actual y_k is obtained by the Powell correction of y_k^ℓ.

2.3 Globalization techniques

To ensure convergence from remote starting points, SQPlab combines a merit function and either linesearch (section 2.3.1) or trust regions (section 2.3.2). Globalization can be desactivated by setting

```
options.algo_globalization = 'unit stepsize';
```

in which case the local method described in section 2.1 applies.

2.3.1 Globalization by linesearch

This is the default globalization option of the solver and it can be ensured by setting (see section 3.1)

```
options.algo_globalization = 'linesearch';
```

With linesearch, the generated search directions need to have a descent property for some merit function. For the while, this property cannot be ensured with Newton's method, so that the algorithm can fail in that case. For the quasi-Newton approach, the descent property can be ensured by various techniques. The simplest one (default) consists in modifying the vector y_k used in the BFGS formula, using *Powell's corrections* as described in section 2.2.2. This is required by setting (see section 3.1)

```
options.algo_descent = 'Powell';
```

If the problem has no constraint or only state constraints, the positive definiteness of the generated matrices can be ensured by using Wolfe's linesearch (or an extension to it). This is required by setting (see section 3.1)

```
options.algo_descent = 'Wolfe';
```

A *merit function* gathers the two aspects of problem (P), minimality of the criterion f and feasibility of the constraints c. In SQPlab, we use the merit function $\Theta_{\mu,\sigma} : \mathbb{R}^n \to \mathbb{R}$ defined at $x \in \mathbb{R}^n$ by

$$\Theta_{\mu,\sigma}(x) = f(x) + \mu^\top c(x) + \sigma \|c(x)^\#\|_1,$$

where μ is an updated *multiplier estimate* (it is nonzero, only when there are state constraints and `options.algo_descent` is set to `'Wolfe'`), $\sigma > 0$ is a penalty parameter, $\|\cdot\|_1$ is the ℓ_1-norm, and the notation $c(x)^\#$ has been defined by (2).

Once the primal-dual solution $(d, \lambda^{\mathrm{QP}})$ to the osculating quadratic problem (8) has been computed, the new iterate (x_+, λ_+) is obtained by

$$x_+ = x + \alpha d \quad \text{and} \quad \lambda_+ = \lambda + \alpha(\lambda^{\mathrm{QP}} - \lambda), \tag{21}$$

where the stepsize $\alpha > 0$ can be set to one (this is convenient in a neighborhood of the solution) or determined by linesearch. The latter forces the decrease of $\Theta_{\mu,\sigma}$.

2.3.2 Globalization by trust regions

Trust region globalization is activated by setting (see section 3.1)

```
options.algo_globalization = 'trust regions';
```

In SQPlab 0.4.4, this option can only be used when the are equality constraints and no other contraints (bound or inequality); hence there must hold $n_B = m_I = m_S = 0$.

3 Usage

The arguments of the procedure `sqplab` are described in section 3.1. In section 3.3, we give the typical sequence of statements that must precede a call to the solver.

3.1 The solver

Here is the form of the `sqplab` procedure:

```
[x,lm,info] = sqplab (@simul, x, lm, lb, ub, options)
```

Input arguments. The first two input arguments are mandatory the other ones are optional. If an optional argument is present, those preceding it must also be present.

`@simul`: is the function handle of the user-supplied simulator. If this one is named `mysimul`, use the handle `@mysimul` as the first argument of `sqplab`. In SQPLAB, the simulator is indeed called by

```
[...]  = simul (...)
```

not by `[...] = feval (simul,...)`. See section 3.2 for more details.

`x`: vector giving the initial guess of the solution to problem (P). The length of `x` is used to determine the number of variables n.

`lm` (optional): vector giving the initial guess of the dual solution (Lagrange or KKT *multiplier*):

- `lm(1:n)` $= \lambda_B^+ - \lambda_B^-$ is the difference between the multiplier λ_B^+ associated with the bound constraint $x \leqslant u_B$ and the multiplier λ_B^- associated with the bound constraint $l_B \leqslant x$,
- `lm(n+1:n+mi)` $= \lambda_I^+ - \lambda_I^-$ is the difference between the multiplier λ_I^+ associated with the inequality constraint $c_I(x) \leqslant u_I$ and the multiplier λ_I^- associated with the inequality constraint $l_I \leqslant c_I(x)$,
- `lm(n+mi+1:n+mi+me)` $= \lambda_E$ is the multiplier associated with the equality constraint $c_E(x) = 0$,
- `lm(n+mi+me+1:n+mi+me+ms)` $= \lambda_S$ is the multiplier associated with the state constraint $c_S(x) = 0$.

The dimensions `mi` $= m_I$, `me` $= m_E$, and `ms` $= m_S$ are known after the first call to the simulator (see section 3.2). It is often difficult to find a good value for the multiplier λ; actually it has the meaning of a *marginal cost*, specifying how the optimal cost varies when the corresponding constraint is perturbed. If you have no idea of that value, just set `lm=[]`, hence letting `sqplab` choose itself the best multiplier.

The default value computed by `sqplab` is the *least-squares multiplier* (this computation is done when `lm` is set to `[]` or is not present as an argument).

`lb` (optional): vector of dimension $n + m_I$ giving the lower bound on x (first n components) and $c_I(x)$. It can have infinite components.

The default value is `-inf` (no lower bound).

`ub` (optional): vector of dimension $n + m_I$ giving the upper bound on x (first n components) and $c_I(x)$. It can have infinite components.

The default value is `inf` (no upper bound).

`options` (optional): structure for tuning the behavior of `sqplab`. In the strings below, the case is meaningless and multiple white spaces are considered as a single white space. The following fields can be used.

- `options.algo_descent` specifies the technique used to ensure descent of the direction solution to the osculating QP.
  - `'Powell'`: assumes that the Hessian of the Lagrangian is approximated by the BFGS formula (hence `algo_method` = `'quasi-Newton'`) and that the positive definiteness of the generated matrices is ensured by Powell's corrections.
  - `'Wolfe'` assumes that the Hessian of the Lagrangian is approximated by the BFGS formula (hence `algo_method` = `'quasi-Newton'`) and that the positive definiteness of the generated matrices is ensured by the Wolfe linesearch.
- `options.algo_globalization` specifies the type of globalization technique to use.
  - `'unit stepsize'` prevents `sqplab` from using a globalization technique. In other words, α in (21) is set to 1.
  - `'linesearch'` (default) requires `sqplab` to force convergence with linesearch or piecewise linesearch.
- `options.algo_method` specifies the second order information used in the algorithm:
  - `'Newton'` requires a Newton algorithm; second order derivatives will be computed either in the form of the Hessian of the Lagrangian L defined by (15)-(16) (when $m_S = 0$) or in the form of the reduced Hessian of the Lagrangian $\bar{L}$ defined by (17) (when $m_S \neq 0$); see section 2.2.1;
  - `'quasi-Newton'` (default) requires a quasi-Newton algorithm; only first order derivatives will be computed; when $m_S = 0$ the full Hessian of the Lagrangian is approximated; when $m_S \neq 0$ (optimal control problems), the reduced Hessian of the Lagrangian is approximated and at least two linearizations of the state constraints are performed at each iteration; see section 2.2.2.
- `options.df1` can be used in unconstrained optimization to specify the expected decrease of the objective function at the first iteration. When it is positive (> 0), this value is used by the quasi-Newton algorithm to scale the initial matrix (using Fletcher's formula), hence avoiding useless stepsize trials at the first iteration. A good value of `options.df1` is often difficult to find, but for least-squares problems the value $f(x_1)/10$ or $f(x_1)/100$ is often adequate. As a rule, a too large value is not dangerous: the stepsize is reduced by SQPLAB in a few internal iterations. On the other hand, an excessively small value of `options.df1` could, due to rounding error, force

SQPlab to stop at the first iteration.

The default value is `0` (unknown expected initial decrease).

- `options.dxmin` is a positive number specifying the precision to which the primal variables must be determined. If `sqplab` needs to make a step smaller than `dxmin` in the infinity-norm to progress to optimality, it will stop. To this respect, a too small value for `dxmin` will force the solver to work for nothing at the very end when rounding errors prevent it from making any progress.

  The value of `dxmin` is also used to detect active bounds (for the bound constraints on x and the bound constraints on $c_I(x)$), so that this value intervenes in the complementarity conditions.

  The default value is `1.e-8`.

- `options.fout` is the file identifier (FID) for the printed outputs.

  The default value is `1`, which implies that the outputs are written on the screen.

- `options.inf` is used to specified infinite (or nonexistent) bounds. A lower bound `lb(i)` $\leqslant -$`options.inf` is considered to be infinitely negative (or nonexistent) and an upper bound `ub(i)` $\geqslant$ `options.inf` is considered to be be infinitely positive (or nonexistent).

  The default value is `inf` (the largest number in Matlab).

- `options.miter` maximum number of iterations.

  The default value is 1000.

- `options.tol` tolerance on optimality:

  – `options.tol(1)` is the tolerance on the gradient of the Lagrangian,
  – `options.tol(2)` is the tolerance on the feasibility,
  – `options.tol(3)` is the tolerance on the complementarity.

  More specifically, as soon as (x, λ) satisfies

$$\begin{aligned} \|\nabla_x \ell(x,\lambda)\|_\infty &\leqslant \texttt{options.tol(1)} \\ \|c(x)^\#\|_\infty &\leqslant \texttt{options.tol(2)} \end{aligned}$$

  and complementarity is satisfied, it is considered to be optimal.

- `options.verbose` is the verbosity level for the outputs:

  $= 0$ nothing is printed; the only manner to be informed of the behavior of `sqplab` is to look at the structure `info` (see below);
  $\geqslant 1$ error messages (default);
  $\geqslant 2$ initial setting and final status;
  $\geqslant 3$ one line per iteration;
  $\geqslant 4$ details on the iterations;
  $\geqslant 5$ details on the step computation and on the globalization;
  $= 6$ some additional information is printed, generally requiring expensive computation, such as the evaluation of the eigenvalues of M.

Output arguments. None of the output arguments must be present. If an output argument is present those preceding it must also be present.

`x`: vector of dimension n giving the computed primal solution x.

`lm`: vector of dimension $m = n+m_I+m_E+m_S$ giving the computed dual solution or *multiplier* λ. See the description of input variable `lm` for the meaning of its components.

`info`: structure providing various information on the minimization realized by `sqplab`. The following fields are meaningful.

- `info.ae` value of the Jacobian of the equality constraint function c_E at the final point x.
- `info.ai` value of the Jacobian of the inequality constraint function c_I at the final point x.
- `info.ce` value of the equality constraint function c_E at the final point x.
- `info.ci` value of the inequality constraint function c_I at the final point x.
- `info.compl` value of the complementarity at the final point (x, λ).
- `info.cs` value of the state constraint function c_S at the final point x.
- `info.f` value of the cost function at the final point x.
- `info.feasn` specifies the ℓ_∞ norm of the feasibility, i.e.,

$$\| \max(0, l - c_{B\cup I}(x), c_{B\cup I}(x) - u)\|_\infty.$$

- `info.flag` specifies the output status of the solver and the computed variables.
  - = 0: a solution has been found up to the required accuracy,
  - = 1: failure because one of the input argument is wrong,
  - = 2: failure because the problem structure is not accepted or some required options are not compatible,
  - = 3: error when running the simulator
  - = 4: stop required by the simulator,
  - = 5: maximum iteration reached,
  - = 6: maximum simulation reached,
  - = 7: stop on `dxmin`,
  - = 8: impossible to decrease the merit function,
  - = 9: the direction d computed by the QP solver is not a descent direction of the merit function,
  - = 10: ill-conditioning,
  - = 20: the solution to the QP is zero,
  - = 21: infeasible QP,
  - = 22: unbounded QP,
  - = 99: strange, such an error should not occur (call your guru).
- `info.g` value of the gradient of the cost function at the final point x.
- `info.glag` provides $\nabla_x \ell(x, \lambda)$, the gradient with respect to x of the Lagrangian ℓ defined by (3),
- `info.glagn` provides $\|\nabla_x \ell(x, \lambda)\|_\infty$, the ℓ_∞ norm of the gradient with respect to x of the Lagrangian defined by (3),

- `info.niter` specifies the realized number of iterations,
- `info.nsimul(i)` specifies the realized number of simulations with `indic` = `i` (see section 3.2 for the meaning of `indic`).

3.2 The simulator

The *simulator* is a user-supplied procedure that evaluates the value of the functions defining (P), as well as its derivatives. `Sqplab` uses the *input indicator argument* `indic` to tell the simulator what it has to compute. In other words, the simulator does not use its number of input and/or output arguments to decide what it has to do, but the value in `indic`. We have found this technique less ambiguous. When the simulator has done the required computation (if this is possible), it sends back in the output variables the result expected by `sqplab`. It also specifies by setting appropriately the *output indicator argument* whether the computation has been realized. By the same `outdic` argument, the simulator can send a message to the optimization procedure, telling, for example that it is desirable to stop at the current point.

Below, the simulator is supposed to be named "`mysimul`". Then, `sqplab` must be called with `simul` set to `@mysimul`. `Sqplab` calls `mysimul` in one of the following five ways.

```
[outdic] = mysimul (indic,x,lm)                          % 1
[outdic,f,ci,ce,cs,g,ai,ae] = mysimul (indic,x)       % 2:4
[outdic,hl] = mysimul (indic,x,lm)                       % 5:6
[outdic,hlv] = mysimul (indic,x,lm,v)                      % 7
[outdic,mv] = mysimul (indic,x,v)                     % 11:16
```

For clarity, we have put on the right hand side the values of `indic` that each statement can accept. We examine each statement in the following sections.

3.2.1 Free call

At the beginning of every iteration, `sqplab` calls the simulator without requiring any computation from it. To indicate this fact, `sqplab` set `indic` to 1. The simulator can take the opportunity of this call to do whatever it makes sense or is useful for it: printing results in some file or plotting them is very standard. Since this call is done at every iteration, the simulator can know the iteration index by counting the iterations, which is sometimes useful. The call statement is the following.

```
[outdic] = mysimul (indic,x,lm)                  % indic = 1
```

Input arguments.

`indic`: scalar variable that is set to 1 by `sqplab` in this case.

`x`, `lm`: specify the current value of the primal-dual iterate $(x, \lambda) = (\mathtt{x}, \mathtt{lm}) \in \mathbb{R}^n \times \mathbb{R}^m$ at the beginning of the iteration.

Output arguments.

The output argument `outdic` need not be present. See section 3.2.2 for its meaning.

3.2.2 Function and first order derivative computations

`Sqplab` can call the simulator to compute either the functions defining the problem (P), or their derivatives, or both functions and derivatives. `Sqplab` calls the simulator before the optimization loop to get the dimensions m_I = `mi` = `length(ci)`, m_E = `me` = `length(ce)`, and m_S = `ms` = `length(cs)`. A zero dimension means the absence of the corresponding constraint.

```
[outdic,f,ci,ce,cs,g,ai,ae] = mysimul (indic,x)
                                        % indic = 2:4
```

Input arguments.

`indic`: scalar variable indicating what the simulator has to compute. Here are the possible values used by `sqplab`.

- = 2: the simulator has to compute `f`, `ci`, `ce`, and `cs`.
- = 3: the simulator has to compute `g`, `ai`, and `ae`, and do what will be needed for the subsequent evaluations of products of a vector with one of the matrices $A_S(x)$, $A_S^-(x)$, $Z_S^-(x)$ and their transpose (see section 3.2.4).
- = 4: this is `indic` = 2 and `indic` = 3 together; in full words, the simulator has to compute `f`, `ci`, `ce`, `cs`, `g`, `ai`, and `ae`, and do what will be needed for the subsequent evaluations of products of a vector with one of the matrices $A_S(x)$, $A_S^-(x)$, $Z_S^-(x)$ and their transpose (see section 3.2.4).

`x`: point x at which the functions and their derivatives have to be computed.

Output arguments. None of the output arguments must be present. If an output argument is present those preceding it must also be present.

`outdic`: scalar variable, which is a message sent by the simulator to the optimization solver. Here are the values that are meaningful for `sqplab`.

- = 0: the required computation has been done.
- = 1: the given `x` = x is out of an implicit domain (the simulator does not want to evaluate functions at that point). If a globalization technique has been required (see the option `options.algo_globalization` in the description of `sqplab`), `sqplab` will find a point closer to the previous accepted iterate (hence the implicit domain must be an open set and "strong" constraints like c_I, c_I, and c_S cannot be taken into account by this technique); otherwise, `sqplab` will stop.
- = 2: the simulator wants to stop and this is what `sqplab` will do.
- = 3: something wrong happened during the simulation. For example, the code corresponding to the given value of `indic` has not been implemented. In that case, `sqplab` will stop.

`f`: the cost function `f` $= f(x)$.

`ci`: the inequality constraint function `ci` $= c_I(x)$.

`ce`: the equality constraint function `ce` $= c_E(x)$.

`cs`: the state constraint function `cs` $= c_S(x)$.

`g`: the gradient `g` $= \nabla f(x)$ of f at x, which is the vector of its partial derivatives.

`ai`: the Jacobian `ai` $= c_I'(x)$ of c_I at x, whose (i, j) element is the partial derivative of c_i $(i \in I)$ with respect to x_j.

`ae`: the Jacobian `ae` $= c_E'(x)$ of c_E at x, whose (i, j) element is the partial derivative of c_i $(i \in E)$ with respect to x_j.

3.2.3 Second order derivative computations

When `options.algo_method` is set to `'Newton'`, `sqplab` needs second derivatives. It can be the Hessian (when $S = \varnothing$), the reduced Hessian (when $S \neq \varnothing$) of the Lagrangian at the successive iterates (x, λ) or the latter right-multiplied by a vector.

Recall that the *Hessian of the Lagrangian* is the symmetric $n \times n$ matrix $L \equiv L(x, \lambda)$ whose (i, j) element is given by (16), while the *reduced Hessian of the Lagrangian* is the symmetric $(n-m_S) \times (n-m_S)$ matrix given by (17), where $Z_S^- \equiv Z_S^-(x)$ is the tangent basis matrix defined in section 1.2.

When `sqplab` needs the Hessian of the Lagrangian (only if $S = \varnothing$), it calls the simulator with `indic` $= 5$, when it needs the reduced Hessian of the Lagrangian (only if $S \neq \varnothing$), it calls the simulator with `indic` $= 6$, and when it needs the Hessian of the Lagrangian times a vector, it calls the simulator with `indic` $= 7$. The simulator is never called with `indic` $= 5$ and `indic` $= 6$ in the same problem optimization, since either $S = \varnothing$ or $S \neq \varnothing$.

```
[outdic,hl] = mysimul (indic,x,lm)          % indic = 5
[outdic,rhl] = mysimul (indic,x,lm)         % indic = 6
[outdic,hlv] = mysimul (indic,x,lm,v)       % indic = 7
```

Input arguments.

`indic`: scalar variable indicating what the simulator has to compute. Here are the possible values corresponding to second order derivatives calculation.

- $= 5$: the simulator has to compute the Hessian of the Lagrangian and to put it in `hl`;
- $= 6$: the simulator has to compute the *reduced* Hessian of the Lagrangian and to put it in `rhl`;
- $= 7$: the simulator has to compute the Hessian of the Lagrangian times the vector `v` $\in \mathbb{R}^n$ and to put it in `hlv`.

`x`, `lm`: specify the point $(x, \lambda) =$ (`x`, `lm`) $\in \mathbb{R}^n \times \mathbb{R}^m$ at which the (reduced) Hessian of the Lagrangian has to be computed.

`v`: vector of dimension n that will right-multiply the Hessian of the Lagrangian when the simulator is called with `indic` $= 7$.

Output arguments.

`outdic`: scalar variable, with the same meaning as in section 3.2.2.

`hl`: will contain the $n \times n$ Hessian of the Lagrangian L when `indic` $= 5$.

`rhl`: will contain the $(n-m_S) \times (n-m_S)$ reduced Hessian of the Lagrangian $Z_S^{-\top} L Z_S^-$ when `indic` $= 6$.

`hlv`: will contain the product Lv when `indic` $= 7$.

3.2.4 Optimal control computations

When the state constraint $c_S(x) = 0$ is present, the simulator must be able to compute the matrix-vector products $A_S(x)v$, $A_S(x)v$, $A_S^-(x)v$, $A_S^{-\top}(x)v$, $Z_S^-(x)v$, and $Z_S^{-\top}(x)v$ introduced in section 1.2. Recall that

- $A_S(x) = c_S'(x)$ is the $m_S \times n$ Jacobian of the state constraints c_S at x,
- $A_S^-(x)$ is an $n \times m_S$ *right inverse* of $A_S(x)$, and
- $Z_S^-(x)$ is an $n \times (n-m_S)$ matrix whose columns form a basis of the null space of $A_S(x)$.

Of course, the matrices $A_S(x)$, $A_S^-(x)$, and $Z_S^-(x)$, or the pieces forming them, should not be recomputed each time a matrix-vector product is required. Indeed, in general, many products are required with various vectors v and the same x. Therefore, the simulator will compute the matrices $A_S(x)$, $A_S^-(x)$, and $Z_S^-(x)$, or the pieces forming them, only once per iteration, when it is called with `indic` $= 3$ or `indic` $= 4$ (see section 3.2.2).

To require the computation of these products from the simulator, `sqplab` uses the following statement.

```
[outdic,mv] = mysimul (indic,x,v)      % indic = 11:16
```

Input arguments.

`indic`: scalar variable that specifies which of the matrix-vector product $A_S(x)v$, $A_S(x)^\top v$, $A_S^-(x)v$, $A_S^{-\top}(x)v$, $Z_S^-(x)v$, or $Z_S^{-\top}(x)v$ to compute.
$= 11$: the computation of $A_S(x)v$ is required.
$= 12$: the computation of $A_S(x)^\top v$ is required.
$= 13$: the computation of $A_S^-(x)v$ is required.
$= 14$: the computation of $A_S^{-\top}(x)v$ is required.
$= 15$: the computation of $Z_S^-(x)v$ is required.
$= 16$: the computation of $Z_S^{-\top}(x)v$ is required.

`x`: point x at which the Jacobian $A_S(x) = c_S'(x)$, its right inverse $A_S^-(x)$, or the basis matrix $Z_S^-(x)$ must be evaluated.

`v`: vector v intervening in the required matrix-vector product and whose dimension depends on the value of `indic`: $v \in \mathbb{R}^n$ if `indic` $= 11$, $v \in \mathbb{R}^{m_S}$ if `indic` $= 12$, $v \in \mathbb{R}^{m_S}$ if `indic` $= 13$, $v \in \mathbb{R}^n$ if `indic` $= 14$, $v \in \mathbb{R}^{n-m_S}$ if `indic` $= 15$, and $v \in \mathbb{R}^n$ if `indic` $= 16$. When `options.algo_method` $=$ `'Newton'` and `indic` $= 16$, v can be a matrix with n rows.

Output arguments.

`outdic`: scalar variable, with the same meaning as in section 3.2.2.

`mv`: matrix-vector product, whose meaning depends on the value of `indic`:

- `mv` $= A_S(x)v$ if `indic` $= 11$,
- `mv` $= A_S(x)^\top v$ if `indic` $= 12$,
- `mv` $= A_S^-(x)v$ if `indic` $= 13$,
- `mv` $= A_S^{-\top}(x)v$ if `indic` $= 14$,
- `mv` $= Z_S^-(x)v$ if `indic` $= 15$,
- `mv` $= Z_S^{-\top}(x)v$ if `indic` $= 16$.

3.3 Calling sequence

Necessarily, the instructions preceding the call to SQPlab must include the following items.

1. Initialization of the primal variable `x` $= x \in \mathbb{R}^n$ and (optionally) the constraint *multiplier* `lm` $= \lambda \in \mathbb{R}^{n+m_I+m_E+m_S}$.
2. Setting the options in `options` (see section 3.1).
3. Calling `sqplab`.

3.4 Other tools

The SQPlab package also offers some other procedures, which are not used by the software, but which can be useful in some simulators. They are described below.

3.4.1 QR factorization

The function `qplab_qrg` realizes the QR *factorization* of a matrix A, using *Givens rotations*: $A = QR$, where Q is an orthogonal matrix and R is upper triangular of the same dimensions as A. The Matlab function `qr` uses *Householder reflections* instead.

```
[Q,R] = sqplab_qrg (A)
```

Input arguments.

`A`: real matrix of any dimensions, say $m \times n$, without any particular properties.

Output arguments.

`Q`: orthogonal matrix of order n.

`R`: Upper triangular matrix of dimension $m \times n$, such that $A = QR$.

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, 4, 6

[2] R.H. Byrd (1987, May). Robust trust region methods for constrained optimization. Third SIAM Conference on Optimization, Houston, TX. 5

[3] J.E. Dennis, R.B. Schnabel (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs. 7

[4] E.O. Omojokun (1991). *Trust region algorithms for optimization with nonlinear equality and inequality constraints*. PhD Thesis, Department of Computer Science, University of Colorado, Boulder, Colorado 80309. 5

[5] M.J.D. Powell (1978). Algorithms for nonlinear constraints that use Lagrangian functions. *Mathematical Programming*, 14, 224–248. 7

Index