SQPlab – A Matlab software for solving nonlinear optimization problems

Version 0.3 (March, 2007)

J. Charles Gilbert[†]

1 The problem to solve **1**
- 1.1 The problem . . . 1
- 1.2 State constraints . . . 2

2 Description of the method **3**
- 2.1 Osculating quadratic problem . . . 3
- 2.2 Hessians and their approximation . . . 4
- 2.3 Globalization . . . 5

3 Usage **5**
- 3.1 The solver . . . 5
- 3.2 The simulator . . . 8
  - 3.2.1 Free call . . . 8
  - 3.2.2 Function and first order derivative computations 9
  - 3.2.3 Second order derivative computations . . . 10
  - 3.2.4 Optimal control computations . . . 11
- 3.3 Calling sequence . . . 12
- 3.4 Other tools . . . 12
  - 3.4.1 QR factorization . . . 12

1 The problem to solve

The name of the software, SQPlab, stands for Sequential Quadratic Programming (SQP) laboratory. It is written in Matlab.

1.1 The problem

SQPlab can solve a general nonlinear optimization problem of the form

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

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. The function $c_S : \mathbb{R}^n \to \mathbb{R}^{m_S}$ imposes additional

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

equality constraints that are treated by SQPLAB in a way that is appropriate to *optimal control problems* (see section 1.2). Smoothness means that at least first order differentiability is required. The notation $l \leq (x, c_I(x)) \leq u$ 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). 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 $m := n + m_I + m_E + m_S$.

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

$$\ell(x, \lambda) = f(x) + \lambda^\top c(x).$$

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 $c_i(x) \leq u_i$ and the one with the lower bound constraint $l_i \leq c_i(x)$. 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:

$$\begin{cases} \text{(a)} & \nabla_x \ell(x, \lambda) = 0 \\ \text{(b)} & l \leq (x, c_I(x)) \leq u, \quad c_E(x) = 0, \quad c_S(x) = 0 \\ \text{(c)} & \forall i \in B \cup I : \quad \lambda_i^-(l_i - c_i(x)) = \lambda_i^+(c_i(x) - u_i) = 0, \end{cases} \tag{3}$$

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

$$\exists \gamma_S > 0, \ \forall x \in \mathcal{X}_S, \ \forall v \in \mathbb{R}^{m_S} : \quad \|A_S(x)^\top v\| \geq \gamma_S \|v\|. \tag{4}$$

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

$$\forall x \in \mathcal{X}_S\text{: } A_S^-(x) \text{ is injective and } A_S(x)A_S^-(x) = I_{m_S}. \tag{5}$$

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:

$$\forall x \in \mathcal{X}_S\text{: } Z_S^-(x) \text{ is injective and } A_S(x)Z_S^-(x) = 0. \tag{6}$$

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 a *osculating quadratic problems* (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)):

$$\begin{cases} \min_{d\in\mathbb{R}^n} \ g^\top d + \frac{1}{2} d^\top M d \\ \tilde{l} \leq (d, A_I d) \leq \tilde{u} \\ c_{E\cup S} + A_{E\cup S} d = 0, \end{cases} \tag{7}$$

where g is the gradient $\nabla f(x)$, M is 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 avoid a QP that, otherwise, would be NP-hard. The osculating QP is then convex. See section 2.2 for more details on the computation of M.

When $S = \varnothing$, SQPLAB solves (7) thanks to the QP solver `quadprog` from the optimization toolbox of Matlab.

When $S \neq \varnothing$, SQPLAB eliminates the linearized state constraints from (7) as follows. Any solution to problem (7) 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

$$d = r + t, \tag{8}$$

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

$$r := -A_S^- c_S \in \mathbb{R}^n, \tag{9}$$

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

$$\begin{cases} \min_{h\in\mathbb{R}^{n-m_S}} \ (g + Mr)^\top Z_S^- h + \frac{1}{2} h^\top Z_S^{-\top} M Z_S^- h \\ \tilde{l}' \leq (Z_S^- h, A_I Z_S^- h) \leq \tilde{u}' \\ c_E + A_E r + A_E Z_S^- h = 0. \end{cases} \tag{10}$$

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*.

2.2 Hessians and their approximation

In the osculating quadratic problem (7), the symmetric $n \times n$ matrix M should be a convenient and good approximation of the *Hessian of the Lagrangian*

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

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

$$L_{ij} = \frac{\partial^2 \ell}{\partial x_i \partial x_j}(x,\lambda). \tag{11}$$

When $M = L$, the SQP algorithm without globalization technique is a Newton-like method, which converges locally quadratically in (x,λ).

When $M = L$ in (10), the $(n-m_S)\times(n-m_S)$ matrix

$$\bar{L} := Z_S^{-\top} L Z_S^- \equiv Z_S^-(x) L(x,\lambda) Z_S^-(x)^\top \tag{12}$$

is called the *reduced Hessian of the Lagrangian*.

The usually expensive matrix Z_S^- intervenes at many places in (10). For some problem, this can be a deterrent factor. Therefore, in the present version of SQPLAB, it is assumed that, when $S \neq \varnothing$, the other constraints does not involve the state variables. This implies that the matrix Z_S^- disappears from the constraints of (10). Then, SQPLAB neglects the term $r^\top M Z_S^- h$ in the criterion of (10) and approximates the reduced Hessian of the Lagrangian $Z_S^{-\top} L Z_S^-$ by a matrix $\bar{M}$ using a quasi-Newton technique. With this assumption and simplifications, the matrix Z_S^- largely disappears from (10). A few matrix-vector multiplications $Z_S^- v$ will still be necessary for computing the reduced gradient $\bar{g} := Z_S^{-\top} g$ and the update of the reduced matrix $\bar{M} \simeq Z_S^{-\top} L Z_S^-$.

This positive definiteness can be ensured by the *BFGS formula* with *Powell's correction*. ...

2.3 Globalization

SQPLAB combines merit function and linesearch to ensure convergence from remote starting points.

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 $\Theta_\sigma : \mathbb{R}^n \to \mathbb{R}$ defined at $x \in \mathbb{R}^n$ by

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

where $\sigma > 0$ 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, λ^{QP}) to the osculating quadratic problem (7) has been computed, the new iterate (x_+, λ_+) is obtained by

$$x_+ = x + \alpha d \quad \text{and} \quad \lambda_+ = \lambda + \alpha(\lambda^{\text{QP}} - \lambda), \tag{13}$$

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 Θ_σ.

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`: string giving the name of the user-supplied simulator; 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 $x \leq u_B$ and the multiplier λ_B^- associated with the bound constraint $l_B \leq x$,
- `lm(n+1:n+mi)` $= \lambda_I^+ - \lambda_I^-$ is the difference between the multiplier λ_I^+ associated with the inequality constraint $c_I(x) \leq u_I$ and the multiplier λ_I^- associated with the inequality constraint $l_I \leq c_I(x)$,
- `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 default value is the least-squares multiplier. 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).

`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`.

`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`.

`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_method` specifies the second order information used in the algorithm:
  - `'quasi-Newton'` for a quasi-Newton technique (hence only with first order derivatives are required),
  - `'Newton'` for a Newton technique (hence second order derivatives are required).
- `options.algo_globalization` specifies the type of globalization technique to use.
  - `'unit stepsize'` prevents `sqplab` to use a globalization technique. In other words, α in (13) is set to 1.
  - `'linesearch'` requires `sqplab` to force convergence with linesearch (default).
- `options.algo_linear_solver` specifies the type of algorithm must be used to solve linear systems.
  - `'direct'`: the linear systems are solved by *direct solvers*, hence using factorization,

– `'cg'`: the linear systems are solved by *conjugate gradient*, hence using factorization,

- `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. Specifying a too small value for `dxmin` will force the solver to work for nothing at the very end when rounding errors prevent making any progress. The default value is `1.e-20`.
- `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)` $\leq -$`options.inf` is considered to be infinitely negative (or nonexistent) and an upper bound `ub(i)` $\geq$ `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.msimul` maximum number of simulations. 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.

  Therefore, as soon as (x, λ) satisfies

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

  it is considered as optimal. The complementarity conditions are also verified.
- `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);
  ≥ 1 error messages (default);
  ≥ 2 initial setting and final status;
  ≥ 3 one line per iteration;
  ≥ 4 details on the iterations;
  ≥ 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 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.flag` specifies the output status.
  - = 0: a solution has been found up to the required accuracy,
  - = 1: failure because one of the input argument is wrong,
  - = 2: error when running the simulator for the first time,
  - = 3: maximum iteration reached,
  - = 4: maximum simulation reached,
  - = 5:
  - = 6: stop on `dxmin`,
  - = 7: infeasible QP,
  - = 8: unbounded QP,
  - = 9: the direction d computed by the QP solver is not a descent direction of the merit function Θ_σ,
  - = 99: strange, such an error should not occur (call your guru).
- `info.niter` specifies the realized number of iterations,
- `info.nsimul` specifies the realized number of simulations.

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 *indicator argument* `indic` to tell the simulator what it has to compute. On its side, the simulator does not use its number of input and/or output arguments to decide what it has to do but the value found in the variable `indic`. Then, the simulator does the required computation (if this is possible) and sends back in the output variables the result expected by `sqplab`. It also specifies in the output argument `outdic` 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'` (with the quotes). `Sqplab` calls `mysimul` in one of the following three ways.

```
[outdic] = mysimul (indic,x,lm)
[outdic,f,ci,ce,cs,g,ai,ae] = mysimul (indic,x)
[outdic,hl] = mysimul (indic,x,lm,v)
[outdic,mv] = mysimul (indic,x,v)
```

We examine them in sequence 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.

```
mysimul (indic, x, lm)
```

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.

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 $m_I = \mathtt{mi} = \mathtt{length(ci)}$, $m_E = \mathtt{me} = \mathtt{length(ce)}$, and $m_S = \mathtt{ms} = \mathtt{length(cs)}$. A zero dimension means the absence of the corresponding constraint.

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

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).

Other values are possible, see below for the other forms on the simulator.

`x`: point x at which the function 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 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) ot 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 (11), while the *reduced Hessian of the Lagrangian* is the symmetric $(n-m_S) \times (n-m_S)$ matrix given by (12), 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,rhlv] = mysimul (indic, x, lm, v)
```

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 `rhlv`;

$= 6$: the simulator has to compute the *reduced* Hessian of the Lagrangian and to put it in `rhlv`; in addition, the simulator will compute what is needed for a subsequent computation of the Hessian of the Lagrangian times a vector (call with `indic` $= 7$);

$= 7$: the simulator has to compute the Hessian of the Lagrangian times the vector `v` $\in \mathbb{R}^n$ and to put it in `rhlv`.

`x`, `lm`: specify the point $(x, \lambda) = (\mathtt{x}, \mathtt{lm}) \in \mathbb{R}^n \times \mathbb{R}^m$ 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.

`rhlv`: will contain the $n \times n$ Hessian of the Lagrangian L when `indic` $= 5$, the $(n-m_S) \times (n-m_S)$ reduced Hessian of the Lagrangian $Z_S^{-\top} L Z_S^-$ when `indic` $= 6$, and 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)
```

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 the constraint multiplier `lm` $= \lambda \in \mathbb{R}^{n+m_I+m_E}$.

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.

Index

BFGS formula, 5

complementarity, 2
conjugate gradient, *see* linear system solver
constraint
    bound, 2
    state, 2

dual variable, *see* multiplier

feasibility, 2

Givens rotation, 12

Hessian
    of the Lagrangian, 4, 10
    reduced – of the Lagrangian, 4, 10
Householder reflection, 12

Lagrangian, 2
linear system solver
    conjugate gradient, 7
    direct, 6

marginal cost, *see* multiplier
Matlab function
    `quadprog`, 4
    `simul`, 8–12
    `sqplab_qrg`, 12
    `sqplab`, **5**–**8**, 12
merit function, 5
multiplier, 2

optimality (proper), 2
optimality conditions, 2

Powell's correction, 5
problem
    (P), 1
    optimal control, 2, **2**–**3**
    quadratic, *see* quadratic problem
procedure, *see* Matlab function

QR factorization, 12
quadratic problem
    osculating, 3
    tangent, 4

reduced
    gradient, 4
    Hessian of the Lagrangian, 4
    matrix, 4
restoration, *see* step
right inverse, **3**, 11

simulator, 8
solution, 2
state, *see* constraint, variable
step
    restoration, 4
    tangent, 4

tangent, *see* quadratic problem, step

uniformly surjective, 2

variable
    control, 3
    state, 3