[WIP] Updated blockSQP user's manual

Based on blockSQP manual by Dennis Janka

Reinhold Wittmann

April 17, 2025

Contents

1. Introduction

blockSQP is a sequential quadratic programming method for finding local solutions of nonlinear, nonconvex optimization problems of the form

$$\min_{x \in \mathbb{R}^n} \varphi(x) \tag{1a}$$

$$\text{s.t. } b_\ell \leq \begin{bmatrix} x \\ c(x) \end{bmatrix} \leq b_u. \tag{1b}$$

It is particularly suited for —but not limited to—problems whose Hessian matrix has block-diagonal structure such as problems arising from direct multiple shooting parameterizations of optimal control or optimum experimental design problems.

blockSQP has been developed around the quadratic programming solver qpOASES [2] to solve the quadratic subproblems. Gradients of the objective and the constraint functions must be supplied by the user. The constraint Jacobian may be given in sparse or dense format. Second derivatives are approximated by a combination of SR1 and BFGS updates. Global convergence is promoted by the filter line search of Waechter and Biegler [5, 6] that can also handle indefinite Hessian approximations.

The method is described in detail in [3, Chapters 6–8]. These chapters are largely self-contained. The notation used throughout this manual is the same as in [3]. A publication [4] is currently under review.

blockSQP is published under the very permissive zlib free software license which should allow you to use the software wherever you need. The full license text can be found at the end of this document.

2. Building blockSQP

blockSQP currently consists of three parts.

- The C++ source files including the C++ interface.
- The python interface.
- The julia interface.

The python and julia interfaces are built separately from the C++ interface and depend on it.

2.1. Building the C++ package

blockSQP is built from the source files in the src directory. The following dependencies need to be linked.

- a BLAS library.

- a QP solver.

Linking a QP solver
In addition to linking the shared/ static binary of a QP solver, the preprocessor flag QPSOLVER_[QP solver name (capital letters)] needs to be set. Currently available are

- `qpOASES` [convex and nonconvex QPs, dense and sparse matrices]
- gurobi [convex QPs, dense and sparse matrices]
- qpalm [convex QPs, poor performance for nonconvex QPs, dense and sparse matrices]

thoug gurobi and qpalm are experimental. `qpOASES` is required if sparse nonconvex QPs are to be solved. In this case qpOASES itself requires a sparse linear solver such as `MA57` from `HSL` or `MUMPS`. See the respective manuals of the QP solvers.

2.2. Interfaces to other languages

Python. The python interface code is located in a separate folder of the same name. It requires the pybind11 library, whose headers need to be in the include path and which itself requires the python headers. The pybind11 documentation recommends using CMake to build the python module. Setting PYTHON_EXECUTABLE to a path to a python interpreter before using find_package(pybind11) allows specifying the python environment to build for. pybind11_DIR can be set to the path to the pybind11 cmake files to ensure the right pybind11 installation is found.

Julia. The julia interface code similarly depends on the libcxxwrap-julia library on the C++ side and the corresponding CxxWrap.jl library on the julia side. Using CMake is again recommended for building.

As an alternative, the julia interface can be built on blockSQP_jll from binaryBuilder.jl via the blockSQP.jl [TODO MAKE AVAILABLE] package.

2.3. Building via CMake

Most QP solvers, as well as the used interface libraries provide CMake project files. A CMakeLists.txt build specification is provided for blockSQP that takes care of most of the above build requirements. In addition, separate CMakeLists.txt templates are provided for each of the C++ package, the python interface and the julia interface.
The default build downloads `MUMPS` [1] to build `qpOASES` (`https://projects.coin-or.org/qpOASES`) with it. It is the users responsibility to ensure they are eligible to download and use `MUMPS`. The user needs to specify whether the python and julia interface should be built via the arguments -DPYTHON_INTERFACE=ON / -DJULIA_INTERFACE=ON. In addition, they can provide the path to the python interpreter that should be built for (-DPYTHON_INTERPRETER=...). A path to the CxxWrap library CMake files needs to be provided to build the julia interface (-DCXXWRAP_PATH=...).
See the `readme.md` for further details.

3. Setting up a problem

A nonlinear programming problem (NLP) of the form (1) is characterized by the following information that must be provided by the user:

- The number of variables, n,
- the number of constraints, m,
- the objective function, $\varphi : \mathbb{R}^n \longrightarrow \mathbb{R}$,
- the constraint function, $c : \mathbb{R}^n \longrightarrow \mathbb{R}^m$,
- and lower and upper bounds for the variables and constraints, b_ℓ and b_u.

In addition, `blockSQP` requires the evaluation of the

- objective gradient, $\nabla\varphi(x) \in \mathbb{R}^n$, and the
- constraint Jacobian, $\nabla c(x) \in \mathbb{R}^{m\times n}$.

Optionally, the following can be provided for optimal performance of `blockSQP`:

- In the case of a block-diagonal Hessian, a partition of the variables x corresponding to the diagonal blocks,
- a function r to compute a point x where a reduced infeasibility can be expected, $r : \mathbb{R}^n \longrightarrow \mathbb{R}^n$,
- a partition of x into free and dependent blocks, e.g. controls and dependent stage states. ($\rightarrow$ Array of `vblocks`, see 3.2.2)

3.1. Sparsity format

The functions `initialize` and `evaluate` can be implemented as sparse or dense, the option `sparse_mode` controls which are used.

In `blockSQP`, we work with the column-compressed storage format (Harwell–Boeing format). There, a sparse matrix is stored as follows:

- an array of nonzero elements `double jacNz[nnz]`, where `nnz` is the number of nonzero elements,
- an array of row indices `int jacIndRow[nnz]` for all nonzero elements, and
- an array of starting indices of the columns `int jacIndCol[nVar+1]`.

For the matrix

$$\begin{pmatrix} 1 & 0 & 7 & 3 \\ 2 & 0 & 0 & 0 \\ 0 & 5 & 0 & 3 \end{pmatrix}$$

the column-compressed format is as follows:

```
nnz=6;
jacNz[0]=1.0;
jacNz[1]=2.0;
jacNz[2]=5.0;
jacNz[3]=7.0;
jacNz[4]=3.0;
jacNz[5]=3.0;

jacIndRow[0]=0;
jacIndRow[1]=1;
jacIndRow[2]=2;
jacIndRow[3]=0;
jacIndRow[4]=0;
jacIndRow[5]=2;

jacIndCol[0]=0;
jacIndRow[1]=2;
jacIndRow[2]=3;
jacIndRow[3]=4;
jacIndRow[4]=6;
```

In `examples/example1.cc`, `initialize` and `evaluate` are implemented both sparse and dense using a generic conversion routine that converts a dense matrix (given as `Matrix`) into a sparse matrix in column-compressed format.

Note that the sparsity pattern is not allowed to change during the optimization. That means you may only omit elements of the constraint Jacobian that are *structurally* zero, i.e., that can never be nonzero regardless of the current value of `xi`. On the other hand, `jacNz` may also contain zero values from time to time, depending on the current value of `xi`.

3.2. The C++ interface

`blockSQP` is written in C++ and uses an object-oriented programming paradigm. The method itself is implemented in a class `SQPmethod`. Furthermore, `blockSQP` provides a basic class `Problemspec` that is used to specify an NLP of the form (1). To solve an NLP, first an instance of `Problemspec` must be passed to an instance of `SQPmethod`. Then, `SQPmethod`'s appropriate methods must be called to start the computation.

In the following, we first describe the `Problemspec` class and how to implement the mathematical entities mentioned above. Afterwards we describe the necessary methods of the `SQPmethod` class that must be called from an appropriate driver routine. Some examples where NLPs are specified using the `Problemspec` class and then passed to `blockSQP` via a C++ driver routine can be found in the `examples/` subdirectory.

3.2.1. Class `Problemspec`

To use the class `Problemspec` to define an NLP, you must implement a derived class, say `MyProblem`, where at least the following are implemented:

1. A constructor,
2. the method `initialize`, for sparse or dense Jacobian,
3. the method `evaluate`, for sparse or dense Jacobian.

`blockSQP` can be used with sparse and dense variants of `qpOASES`. Depending on the preferred version (set by the algorithmic option `sparse_mode`, see Sec. 4.7), the constraint Jacobian must be provided in sparse or dense format by the user.

Before passing an instance of `MyProblem` to `blockSQP`, the following attributes must be set:

1. `int nVar` - the number of variables,
2. `int nCon` - the number of constraints (linear and nonlinear),
3. `int nnz` (optional) - number of nonzeros in sparse constraint Jacobian, not required for dense mode,
4. `Matrix lb_var, ub_var` - lower and upper bounds for variables,
5. `Matrix lb_con, ub_con` - lower and upper bounds for constraints,
6. `int nBlocks` - the number of diagonal blocks in the Hessian,
7. `int* blockIdx` - an array of dimension `nBlocks+1` with the indices of the partition of the variables that correspond to the diagonal blocks. It is required that `blockIdx[0]=0` and `blockIdx[nBlocks]=nVar`,
8. `vblock *vblocks` (optional) - an array of `vblock {int size, bool dependent}` (3.2.2) to distinguish free and dependent variables.

An bound is specified by setting it to $\pm$ inf where inf is the value of the option inf, `std::numeric_limits<double>::infinity()` by default.

`Matrix`

The class `Matrix` is a simple interface to facilitate maintaining dense matrices, including access to the individual elements (internally stored column major as an array of `double`). It has the constructor `Matrix(m,n=1,ldim=-1)` for constructing an $m \times n$ `Matrix`, elements can be accessed via operator `(i, j)`.

Function `initialize`

`initialize` is called once by `blockSQP` before the SQP method is started. The dense version takes the following arguments:

- `Matrix &xi`, the optimization variables
- `Matrix &lambda`, the Lagrange multipliers
- `Matrix &constrJac`, the (dense) constraint Jacobian

All variables are initialized by zero on input and should be set to the desired starting values on return. In particular, you may set parts of the Jacobian that correspond to purely linear constraints (i.e., that stay constant during optimization) here.

The sparse version of `initialize` takes the following arguments:

- `Matrix &xi`, the optimization variables
- `Matrix &lambda`, the Lagrange multipliers
- `double *jacNz`, nonzero elements of constraint Jacobian
- `int *jacIndRow`, row indices of nonzero elements
- `int *jacIndCol`, starting indices of columns

`xi` and `lambda` are initialized by zero and must be set the same as in the dense case. `jacNz`, `jacIndRow`,`jacIndCol` are the sparse Jacobian in CCS format 3.1 and already allocated. `jacIndRow` and `jacIndCol` need to be initialized here and one can initialize parts of `jacNz` that stay constant. As such, the sparsity structure cannot be changed during the iterations and therefore should include all structurally nonzero elements.

Function `evaluate`

Similar to `initialize`, two versions of `evaluate` exist. `evaluate` is called repeatedly by `blockSQP` to evaluate functions and/or derivatives for different `xi` and `lambda`. The dense version takes the following arguments:

- `const Matrix &xi`, current value of the optimization variables (input)
- `const Matrix &lambda`, current value of the Lagrange multipliers (input)
- `double *objval`, pointer to objective function value (output)
- `Matrix &constr`, constraint function values (output)
- `Matrix &gradObj`, gradient of objective (output)
- `Matrix &constrJac`, dense constraint Jacobian (output)
- `SymMatrix *hess`, (blockwise) Hessian of the Lagrangian (output)
- `int dmode`, derivative mode (input)
- `int *info`, error flag (output)

Depending on the value of `dmode`, the following must be provided by the user:

- `dmode=0`: compute function values `objval` and `constr`
- `dmode=1`: compute function values and first derivatives `gradObj` and `constrJac`
- `dmode=2`: compute function values, first derivatives, and Hessian of the Lagrangian for the *last*[1] diagonal block, i.e., `hess[nBlocks-1]`
- `dmode=3`: compute function values, first derivatives, and all blocks of the Hessian of the Lagrangian, i.e., `hess[0]`,...,`hess[nBlocks-1]`.

`dmode=2` and `dmode=3` are only relevant if the option `exact_hess_usage` is set to `1` (last block) or `2` (full Hessian). The default is `0`. On return, the variable `info` must be set to `0` if the evaluation was successful and to a value other that `0` if the computation was not successful.

In the sparse version of `evaluate`, the Jacobian must be evaluated in CCS format instead using the arrays `jacNz`, `jacIndRow`, and `jacIndCol` as described in 3.1. Here only `jacNz` should be set.

Function `reduceConstrVio`

Whenever `blockSQP` encounters an infeasible QP or cannot find a step length that provides sufficient reduction in the constraint violation or the objective, it resorts to a feasibility restoration phase to find a point where the constraint violation is smaller. This is usually achieved by solving an NLP to reduce some norm of the constraint violation. In `blockSQP`, a minimum ℓ_2-norm restoration phase is implemented. The restoration phase is usually very expensive: one iteration for the minimum norm NLP is usually more expensive than one iteration for the original NLP! As an alternative, `blockSQP` provides the opportunity to implement a problem-specific restoration heuristic because sometimes a problem "knows" (or has a pretty good idea of) how to reduce its infeasibility[2]

This routine is of course highly problem-dependent. If you are not sure what to do here, just do not implement this method. Otherwise, the method just takes `xi`, the current value of the (infeasible) point as input and expects it to be mutated to a new point. In addition, it takes a flag `info` that must be set indicating if the evaluation was successful, in which case `info=0`.

---

[1] `whichSecondDerv=1` can be useful in a multiple shooting setting: There, the lower right block in the Hessian of the Lagrangian corresponds to the Hessian of the *objective*. See [3] how to exploit this for problems of nonlinear optimum experimental design.

[2] A prominent example are dynamic optimization problems parameterized by multiple shooting: there, additional continuity constraints for the differential states are introduced that can be violated during the optimization. Whenever `blockSQP` calls for the restoration phase, the problem can instead try to integrate all states over the *whole* time interval and set the shooting variables such that the violation due to continuity constraints is zero. This is often enough to provide a sufficiently feasible point and the SQP iterations can continue.

3.2.2. Class Condenser

[LIKELY SUBJECT TO CHANGE]
The condenser class of blockSQP requires the following structs.

- `vblock: {int size, bool dependent}` corresponds to a block of variables and has a flag to mark dependent variables.
- `cblock: {int size, bool removed}` corresponds to a block of constraints and has a flag to mark used-up conditions.
- `condensing_target: {int n_stages, int first_free, int vblock_end, int first_cond, int cblock_end}` describes a target structure for condensing.

Variables and constraints are partitioned into `vblocks` and `cblocks`, with both being given as arrays `vblocks = vblock*, cblocks = cblock*` of lengths `n_vblocks, n_cblocks`. Multiple shooting leads to a sequence of alternating free and dependend variable blocks. The continuity conditions are a sequence of `cblocks` with sizes corresponding the the total size of one or several consecutive dependent variable blocks.
IMPORTANT: Right now, the continuity conditions are expected to be implemented in the form $x^{(k+1)} - S(x^{(k)}, u^{(k)}) = 0$. Support for conditions $S(x^{(k)}, u^{(k)}) - x^{(k+1)}$ may be added in the future.

Each `condensing_target` contains start and end indices of the alternating free-dependent variable blocks and `cblocks` corresponding to the respective continuity conditions.

- `n_stages` - number of shooting stages
- `first_free` - index of first free variable block that dependent blocks depend on
- `vblock_end` - index of the first variables block not dependent or depended upon
- `first_cond` - index of the constraint block that
- `cblock_end` - index of the constraint block that

The condenser class requires the vblocks, cblocks and targets for construction, in addition to an integer array of the Hessian block sizes.

- `vblock *vblocks, int num_vblocks`
- `cblock *cblocks, int num_cblocks`
- `int *hsizes, int n_hsizes`
- `condensing_target *targets, int num_targets`
- `int add_dep_bounds`

The final argument add_dep_bounds determines whether bounds of dependent variables are added as constraints to the condensed QP: 0 - not added, 1 - added, but bounds set to $\pm\infty$, 2 (default) - added.

The method full_condense can then be called to condense a QP. It takes 7 arguments and 7 equivalent return arguments. They are

- Matrix& grad_obj - linear term in the QP
- Sparse_Matrix& constr_jac - the sparse constraint Jacobian (see blocksqp_matrix.hpp)
- SymMatrix* hess - the array of Hessian blocks
- Matrix& lb_var, ub_var, lb_con, ub_con - variable and constraint bounds

This also stores the data necessary to recover the original solution of the specific QP given. The user is expected to solve the condensed QP, then call recover_var_mult with the primal-dual condensed QP solution (Matrix xi_cond, lambda_cond) and the original QP solution (Matrix xi, lambda) as return arguments.

3.2.3. Class SQPmethod and SCQPmethod

If you have implemented a problem using the Problemspec class you may solve it with blockSQP using a suitable driver program. There, you must include the header file blocksqp_method.hpp (and of course any other header files that you used to specify your problem). An instance of SQPmethod is created with a constructor that takes the following arguments:

- Problemspec *problem, the NLP, see above
- SQPoptions *parameters, an object in which all algorithmic options and parameters are stored
- SQPstats *statistics, an object that records certain statistics during the optimization and – if desired – outputs some of them in files in a specified directory

In addition, the SCQPmethod (Sequential Condensed Quadratic Programming) subclass takes an instance of Condenser for the given problem as fourth argument.

Instances of the classes SQPoptions and SQPstats must be created before.
SQPoptions uses the default constructor, its fields are detailed below.

SQPstats takes a char* that describes the directory where output files are created if file output is enabled (SQPoptions::debug_level).

Once an SQPmethodhas been created, the NLP can be solved by calling the following.

- `SQPmethod::init()`: Must be called before . Therein, the user-defined `initialize` method of the `Problemspec` class is called.
- `SQPmethod::run(int maxIt, int warmStart = 0)`: Run the SQP algorithm with the given options for at most `maxIt` iterations. You may call with `warmStart=1` to continue the iterations from an earlier call. In particular, the existing Hessian information is re-used. That means that

  ```
  SQPmethod* method;
  [...]
  method->run( 2 );
  ```

  and

  ```
  SQPmethod* method;
  [...]
  method->run( 1 );
  method->run( 1, 1 );
  ```

  yield the same result. It returns an instance of the `SQPresult` enum class to indicate success (>0), max iteration reached (=0) or failure (<0). See `blocksqp_defs.hpp` for all values of `SQPresult`.
- `SQPmethod::finish()`: Should be called after the last call to `run` to make sure all output files are closed properly.

Again, we strongly recommend to study the example in `examples/example1.cpp`, where all steps are implemented for a simple NLP with block-diagonal Hessian.

3.2.4. Retrieving the primal-dual iterate

The primal iterate can be retrieved by calling

```
Matrix SQPmethod::get_xi()
```

or

```
void SQPmethod::get_xi(Matrix &xi_hold)
```

The Lagrange multipliers/dual iterate is retrieved the same way via the methods

```
Matrix SQPmethod::get_lambda()
void SQPmethod::get_lambda(Matrix &lambda_hold)
```

Important: `blockSQP` defines the Lagrangian as

$$L(\xi,\lambda) = f(\xi) - \lambda^T g(\xi). \tag{2}$$

Therefore, strongly active lower bounds are characterized by the associated Lagrange multiplier being > 0, strongly active upper bounds by the multiplier being < 0. Compared to optimizers defining the Lagrangian as $L(\xi,\lambda) = f(\xi) + \lambda^T g(\xi)$, e.g. `ipopt`[7], the multipliers `blockSQP` returns will have the opposite sign.

3.3. The python interface

The python interface consists of

- The compiled py_blockSQP.so (py_blockSQP.cpython-... .so).
- The blockSQP.py python-side Problemspec class.

They both need to be imported, either directly or through a python __init__.py module specification.

```
import numpy as np
import py_blockSQP
from blockSQP_pyProblem import blockSQP_pyProblem as Problemspec
```

Like in the C++ interface, problem specification, options and statistics objects need to be created and passes to an SQPmethod.

```
opts = py_blockSQP.SQPoptions()
opts.opttol = 1e-5
...
stats = py_blockSQP.SQPstats("./solver/output/path")
...
prob = Problemspec() \# blockSQP\_pyProblem()
```

The following data and function attributes need to be provided.

- nVar [int] - number of optimization variables
- nCon [int] - number of constraints
- f [np.ndarray[np.float64](nVar,) $\rightarrow$ float/np.float64/Cdouble] - objective function
- grad_f [np.ndarray[np.float64]](nVar,) $\rightarrow$ np.ndarray[np.float64](nVar,) - objective gradient
- g [np.ndarray[np.float64]](nVar,) $\rightarrow$ np.ndarray[np.float64](nCon,) - constraint function
- jac_g [np.ndarray[np.float64]](nVar,) $\rightarrow$ np.ndarray[np.float64](nCon, nVar) - constraint Jacobian, only used and required in dense mode

If sparse derivatives should be used, the option sparseQP must be set > 0 and the make_sparse method has to be called with the following arguments

- jac_g_nnz [float] - the number of structural nonzero in the constraint Jacobian
- jac_g_row [np.ndarray[np.int32], shape = (jac_g_nnz,)] - row indices of constraint Jacobian in CCS format
- jac_g_colind [np.ndarray[np.int32], shape = (nVar + 1,)] - column start indices of constraint Jacobian in CCS format.

Finally, the bounds and Hessian block indices should be set by calling the set_bounds method with

- lb_var [np.ndarray[np.float64](nVar,)] - lower variable bounds

- ub_var [np.ndarray[np.float64](nVar,)] - upper variable bounds
- lb_con np.ndarray[np.float64](nCar,)] - lower constraint bounds
- ub_con np.ndarray[np.float64](nCar,)] -upper constraint bounds

and set_blockIndex with hessBlock_index [np.ndarray[np.int32]]. A feasibility restoration heuristic can be added as the attribute `continuity_restoration` mutates an `np.ndarray` of length `nVar`.

Important: `prob.complete()` must be called once all data has been provided to the problem to finalize it.

`blockSQP` is then called as in the C++ interface

```
optimizer = py_blockSQP.SQPmethod(prob, opts, stats)
optimizer.init()
ret = optimizer.run(max_num_iterations)
optimizer.finish()
```

The optimizer returns an SQPresult enum class instance to indicate the result of the optimization, as in the C++ interface. It can be converted either to a string to yield 'SQPresult.[name]' or to an integer. The optimal primal and dual variables and be retrieved via

```
xi_opt = np.array(optimizer.get_xi()).reshape(-1)
lam_opt = np.array(optimizer.get_lambda()).reshape(-1)
```

Important: Mind the sign of the returned multipliers, see Section 3.2.4.

3.4. The julia interface

The julia interface requires the compiled jul_blockSQP.so binary and the julia side module blockSQP.jl. It similar to the Python interface and accessed in the following way

```
using blockSQP


prob = blockSQP.jlProblem(Int32(2), Int32(1))
prob.f = ...

opts = Dict()
opts["..."] = ...

stats = blockSQP.SQPstats("...")

method = blockSQP.Solver(prob, opts, stats)

blockSQP.init!(meth)
blockSQP.run!(!, Int32(...), Int32(...))
blockSQP.finish!(meth)

x_opt = blockSQP.get_primal_solution(meth)
lam_opt = blockSQP.get_dual_solution(meth)
```

Important: Mind the sign of the returned multipliers, see Section 3.2.4.

[WIP] In the future, `blockSQP` may also be called through `Optimization.jl`.

4. Options and parameters

In this section we describe the most important options that are passed to `blockSQP` through the `SQPoptions` class. For the remaining options look into the options.hpp header file.

4.1. Basic options

The most important options for users wanting to solve structured problems.

Name	Description/possible values	Default
`sparse_mode`	Use dense/sparse callbacks	False
`exact_hess_usage`	Request exact Hessian in callbacks 0: Disabled 1: Request last block 2: Request all blocks	0
`hess_approximation`	 0: Scaled identity 1: SR1 2: damped BFGS	1
`fallback_approximation`	See hess_approximation, has to be positive definite	2
`block_hess`	enable blockwise updates 0: Full space updates 1: Partitioned updates* 2: 2 blocks - first n-1 together and last block* * requires providing `blockIdx`	1
`qpsol`	Which QP solver should be used `C++:` passed as `QPsolvers::qpOASES` `Python/Julia:` passed as "qpOASES"	`qpOASES`
`qpsol_options`	Options to be passed to the QP solver `C++:` passed as class `qpOASES_options` `Python/Julia:` passed as a dict	

4.2. qpsol_options

Options to be passed to the QP solver. For `qpOASES` :

Name	Description/possible values	Default

`sparsityLevel`	Which `qpOASES` solver class -1: Infer for `SQPoptions` 0: `qpOASES::SQProblem`, dense matrices 1: `qpOASES::SQProblem`, sparse matrices 2: `qpOASES::SQProblemSchur`, sparse matrices	-1
→ **`qpOASES` manual**		
`printLevel`	Output level of qpOASES	0
`terminationTolerance`		1e-10

See options.hpp for other QP solver's options.

4.3. Common algorithmic options

Name	Symbol/Meaning	Default
`opt_tol`	ε_{opt}	1.0e-6
`feas_tol`	$\varepsilon_{\text{feas}}$	1.0e-6
`eps`	machine precision	1.0e-16
`inf`	∞	`double(∞)`
`max_QP_it`	Maximum number of QP iterations per SQP iteration	5000
`max_QP_seconds`	Maximum time in seconds for `qpOASES` per SQP iteration	10000.0

4.4. Convexification strategy

Regularizing indefinite SR1 or exact Hessians increases the time per SQP iteration, but tends to make the method more consistent. The are quite a few examples that rarely converge with SR1-BFGS but converge fast with SR1 Hessians regularized with scaled identities. The following options configure these `convexification strategies`.

Name	Description/possible values	Default
`conv_strategy`	Type of convexification strategy 0: Convex combination between Hessian and fallback 1: Add scaled identities 2: Add scaled identities for free indices* *requires providing `vblocks`	1
`max_conv_QPs`	Maximum number additional QPs per SQP iteration 1: Hessian - fallback > 1: Hessian - convexified Hessians - fallback	1

Advanced options		
conv_tau_H	Tolerance for convexified step acceptance (see paper)	2/3
conv_kappa_0	Initial scale for added identities (see paper [])	1/16
conv_kappa_max	Maximum scale for added identities	2.0

4.5. Output settings

Name	Description/possible values	Default
print_level	Amount of onscreen output per iteration 0: no output 1: normal output 2: verbose output	1
result_print_color	Control output of result at termination 0: no output 1: no output color 2: colored output	2
debug_level	Amount of file output per iteration 0: no debug output 1: print one line per iteration to file 2: extensive debug output to files (impairs performance)	0

4.6. Sizing strategies

Oren-Luenberger [], Shanno-Phua [] and centered Oren-Luenberger [] strategies are available for sizing Hessian approximations.

Name	Description/possible values	Default
initial_hess_scale	Initial scaling of the Hessian approximation	1.0
sizing_strategy	Hessian sizing strategy 0: off 1: Shanno-Phua 2: Oren-Luenberger (OL) 3: geometric mean of 1 and 2 4: centered Oren-Luenberger (COL)	2
fallback_sizing_strategy	fallback Hessian sizing strategy	4
Advanced options		
COL_eps	COL ε	0.1
COL_tau_1	COL τ_1	0.5
COL_tau_2	COL τ_2	10^4

OL_eps	Oren-Luenberger sizing ε	10^{-4}

4.7. Filter line search

Name	Description	Default
enable_linesearch	Enable filter line search	true
max_linesearch_steps	Max number of step reductions	10
max_consec_reduced_steps	Reset Hessian after # reduced steps	8
max_consec_skipped_updates	Reset Hessian after # skipped updates	100
skip_first_linesearch	Skip line search on first iteration	false

4.8. Advanced termination and line search heuristics

Name	Description	Default
enable_premature_termination	Allow early termination with partial success	false
max_extra_steps	Max steps after reaching tolerance to refine solution	0
max_filter_overrides	Number of times filter rules can be overridden	2

5. Output

When the algorithm is run, it typically produces one line of output for every iteration. The columns of the output are:

Column	Description
it	Number of iteration
qpIt	Number of QP iterations for the QP that yielded the accepted step
qpIt2	Number of QP iterations for the QPs whose solution was rejected
obj	Value of objective
feas	Infeasibility
opt	Optimality
|lgrd|	Maximum norm of Lagrangian gradient
|stp|	Maximum norm of step in primal variables
|lstp|	Maximum norm of step in dual variables
alpha	Steplength
nSOCS	Number of second-order correction steps
sk	Number of Hessian blocks where the update has been skipped
da	Number of Hessian blocks where the update has been damped
sca	Value of sizing factor, averaged over all blocks
QPr	Number of QPs whose solution was rejected

6. Extending `blockSQP`

6.1. New methods

Due to blockSQPs object oriented design, it is possible to extend the solver by subclassing `SQPmethod`. The `SCQPmethod` is such an example that enhances `blockSQP` by a QP condensing step. Further extensions by adding function calls to the main loop and adding more fields to the `SQPoptions` class. `blockSQP` is actively being developed and is continuously made more fine-grained to enhance modularity and extensibility.

6.2. Other QP solvers

QP solvers are added to `blockSQP` by subclassing the abstract `QPsolver` class and filling out its methods. The code should be included based on a preprocessor definition named `QPSOLVER_[name of QP solver]` such as the existing `QPSOLVER_qpOASES`. In addition, the classes defined in `options.h` need to be extended to account for the newly available QP solver. The name of the solver should be added to the enum class `QPsolvers` and a `QPsolver_options` subclass should be created to enable passing options to the QP solver. The interfaces to other languages need to updated consistent with the handling of existing QP solvers.

References

[1] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. Mumps: a general purpose distributed memory sparse solver. In *International Workshop on Applied Parallel Computing*, pages 121–130. Springer, 2000.

[2] Hans Joachim Ferreau, Christian Kirches, Andreas Potschka, Hans Georg Bock, and Moritz Diehl. qpOASES: A parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation*, pages 1–37, 2014.

[3] Dennis Janka. *Sequential quadratic programming with indefinite Hessian approximations for nonlinear optimum experimental design for parameter estimation in differential–algebraic equations*. PhD thesis, Ruprecht-Karls-Universität Heidelberg, 2015. Available at `http://archiv.ub.uni-heidelberg.de/volltextserver/19170/`.

[4] Dennis Janka, Christian Kirches, Sebastian Sager, and Andreas Wächter. An SR1/BFGS SQP algorithm for nonconvex nonlinear programs with block-diagonal Hessian matrix. *submitted to Mathematical Programming Computation*, 2015.

[5] Andreas Wächter and Lorenz T Biegler. Line search filter methods for nonlinear programming: Local convergence. *SIAM Journal on Optimization*, 16(1):32–48, 2005.

[6] Andreas Wächter and Lorenz T Biegler. Line search filter methods for nonlinear programming: Motivation and global convergence. *SIAM Journal on Optimization*, 16(1):1–31, 2005.

[7] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106:25–57, 2006.

A. License

This is the full license text (zlib license):


```
blockSQP -- Sequential quadratic programming for problems with
            block-diagonal Hessian matrix.
Copyright (c) 2012-2015 Dennis Janka <dennis.janka@iwr.uni-heidelberg.de>

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any
damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute
it freely, subject to the following restrictions:

    1. The origin of this software must not be misrepresented;
    you must not claim that you wrote the original software.
    If you use this software in a product, an acknowledgment in the
    product documentation would be appreciated but is not required.

    2. Altered source versions must be plainly marked as such,
    and must not be misrepresented as being the original software.

    3. This notice may not be removed or altered from any source
    distribution.
```