[WIP] Updated blockSQP user's manual

Based on blockSQP manual by Dennis Janka

Reinhold Wittmann

April 2, 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 [6, 7] 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 and 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 allowed for better convergence. qpOASES itself requires a sparse linear solver such as `MA57` from `HSL` or `MUMPS`.

2.2. Building the python and julia interfaces

The python interface code is located in a separate folder of the same name. It requires the pybind11 library, whose headers needs to be in the include path and which itself requires the python headers.

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.

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, independent unfinished CMakeLists.txt 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 wether the python and julia interface should be built. In addition, they can provide the path to the python interpreter that should be built for. A path to the CxxWrap library CMake files needs to be provided to build the julia interface.
→ readme.md

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, $r : \mathbb{R}^n \longrightarrow \mathbb{R}^n$.

3.1. Sparsity format

The functions `initialize` and `evaluate` can be implemented as sparse or dense, depending which variant of `qpOASES` should be used later. For maximum flexibility (i.e. if you want to try both sparse and dense variants of `qpOASES`), both the sparse and the dense versions of `initialize` and `evaluate` should be implemented.

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 sparseQP, see Sec. 4.1), 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. Matrix bl, lower bounds for variables and constraints,
4. Matrix bu, upper bounds for variables and constraints (equalities are specified by setting the corresponding lower and upper bounds to the same values),
5. int nBlocks, the number of diagonal blocks in the Hessian,
6. 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.

The class Matrix is a simple interface to facilitate maintaining dense matrices, including access to the individual elements (internally stored columnwise as an array of double), see the documentation within the source code. We strongly recommend to check out examples/example1.cc for an example implementation of a generic NLP with block structure.

Of course a derived class MyProblem may contain many more methods and attributes to represent special classes of NLPs. An example is the software package muse [3] (part of VPLAN [5]), where derived classes of ProblemSpec are used to represent NLPs that arise from the parameterization of optimal control problems and optimum experimental design problems with multiple shooting or single shooting. There, the derived classes contain detailed information about the structure of constraints and variables, methods to integrate the dynamic states and so on.

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. An important difference to the dense version is the constraint Jacobian: the pointers `jacNz`, `jacIndRow`, and `jacIndCol` that represent the Jacobian in column-compressed format are initialized by `NULL`, the null-pointer. **They must be allocated within `initialize` using C++'s `new` operator!**[1] The memory is freed later by `blockSQP`, so the user does not need to take care of it. Of course you may also set parts of the constraint Jacobian that correspond to purely linear constraints here.

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`

[1] The allocation is not done within `blockSQP` directly because `blockSQP` does not know the number of nonzero elements of the Jacobian a priori. That means a separate call would be required to first find out the number of nonzero elements and then – after allocating the sparse Jacobian – another call to `initialize` to set the linear parts of the Jacobian.

- dmode=2: compute function values, first derivatives, and Hessian of the Lagrangian for the *last*[2] 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] (*not fully supported yet*).

dmode=2 and dmode=3 are only relevant if the option whichSecondDerv 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 sparse format using the arrays jacNz, jacIndRow, and jacIndCol as described above. Note that all these arrays are assumed to be allocated by a call to initialize earlier, their dimension must not be changed!

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[3]

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 a new point xi as output. A flag info must be set indicating if the evaluation was successful, in which case info=0.

3.2.2. Class Condenser

TODO

3.2.3. Class SQPmethodand 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

[2] 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.

[3] 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.

(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

Instances of the classes `SQPoptions` and `SQPstats` must be created before. See the documentation inside the respective header files how to create them.

To solve an NLP with `blockSQP`, call the following three methods of `SQPmethod`:

- `init()`: Must be called before . Therein, the user-defined `initialize` method of the `ProblemSpec` class is called.
- `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.
- `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.cc`, where all steps are implemented for a simple NLP with block-diagonal Hessian.

3.3. The python interface

Both the binary and the pythonside problem specification need to be imported.

```
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,) → float/np.float64/Cdouble] - objective function
- grad_f [np.ndarray[np.float64]](nVar,) → np.ndarray[np.float64](nVar,) - objective gradient
- g [np.ndarray[np.float64]](nVar,) → np.ndarray[np.float64](nCon,) - constraint function
- jac_g [np.ndarray[np.float64]](nVar,) → 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](jac_g_nnz,)] - row indices of constraint Jacobian in CCS format
- jac_g_colind [np.ndarray[np.int32]](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 [np.ndarray[np.float64]](nVar,) → np.ndarray[np.float64](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 object to indicate the result of the optimization, as in the C++ interface. They 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)
```

3.4. The julia interface

TODO

4. Options and parameters

In this section we describe all options that are passed to `blockSQP` through the `SQPoptions` class. We distinguish between algorithmic options and algorithmic parameters. The former are used to choose between different algorithmic alternatives, e.g., different Hessian approximations, while the latter define internal algorithmic constants. As a rule of thumb, whenever you are experiencing convergence problems with `blockSQP`, you should try different algorithmic options first before changing algorithmic parameters.

Additionally, the output can be controlled with the following options:

Name	Description/possible values	Default
`printLevel`	Amount of onscreen output per iteration 0: no output 1: normal output 2: verbose output	1
`printColor`	Enable/disable colored terminal output 0: no color 1: colored output in terminal	1
`debugLevel`	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

4.1. List of algorithmic options

Name	Description/possible values	Default

sparseQP	qpOASES flavor 0: dense matrices, dense factorization of red. Hessian 1: sparse matrices, dense factorization of red. Hessian 2: sparse matrices, Schur complement approach	2
globalization	Globalization strategy 0: full step 1: filter line search globalization	1
skipFirstGlobalization	0: deactivate globalization for the first iteration 1: normal globalization strategy in the first iteration	1
restoreFeas	Feasibility restoration phase 0: no feasibility restoration phase 1: minimum norm feasibility restoration phase	1
hessUpdate	Choice of first Hessian approximation 0: constant, scaled diagonal matrix 1: SR1 2: BFGS 3: [not used] 4: finite difference approximation	1
hessScaling	Choice of scaling/sizing strategy for first Hessian 0: no scaling 1: scale initial diagonal Hessian with σ_{SP} 2: scale initial diagonal Hessian with σ_{OL} 3: scale initial diagonal Hessian with σ_{Mean} 4: scale Hessian in every iteration with σ_{COL}	2
fallbackUpdate	Choice of fallback Hessian approximation (see hessUpdate)	2
fallbackScaling	Choice of scaling/sizing strategy for fallback Hessian (see hessScaling)	4
hessLimMem	0: full-memory approximation 1: limited-memory approximation	1
blockHess	Enable/disable blockwise Hessian approximation 0: full Hessian approximation 1: blockwise Hessian approximation	1
whichSecondDerv	User-provided second derivatives 0: none 1: for the last block 2: for all blocks (same as hessUpdate=4)	0
maxConvQP	Maximum number of convexified QPs (int>0)	1
convStrategy	Choice of convexification strategy 0: Convex combination between hessUpdate and fallbackUpdate	0

	1: Add multiples of identity to first Hessian
	2: Add multiple of identity to free indices of first Hessian

4.2. List of algorithmic parameters

Name	Symbol/Meaning	Default
`opttol`	ε_{opt}	1.0e-5
`nlinfeastol`	$\varepsilon_{\text{feas}}$	1.0e-5
`eps`	machine precision	1.0e-16
`inf`	∞	1.0e20
`maxItQP`	Maximum number of QP iterations per SQP iteration (`int`>0)	5000
`maxTimeQP`	Maximum time in second for `qpOASES` per SQP iteration (`double`>0)	10000.0
`maxConsecSkippedUpdates`	Maximum number of skipped updates before Hessian is reset (`int`>0)	100
`maxLineSearch`	Maximum number of line search iterations (int>0)	20
`maxConsecReducedSteps`	Maximum number of reduced steps before restoration phase is invoked (int>0)	100
`hessMemsize`	Size of Hessian memory (`int`>0)	20
`maxSOCiter`	Maximum number of second-order correction steps	3

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

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] S. Körkel. *Numerische Methoden für Optimale Versuchsplanungsprobleme bei nichtlinearen DAE-Modellen*. PhD thesis, Universität Heidelberg, Heidelberg, 2002.

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

[7] 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.

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.
```