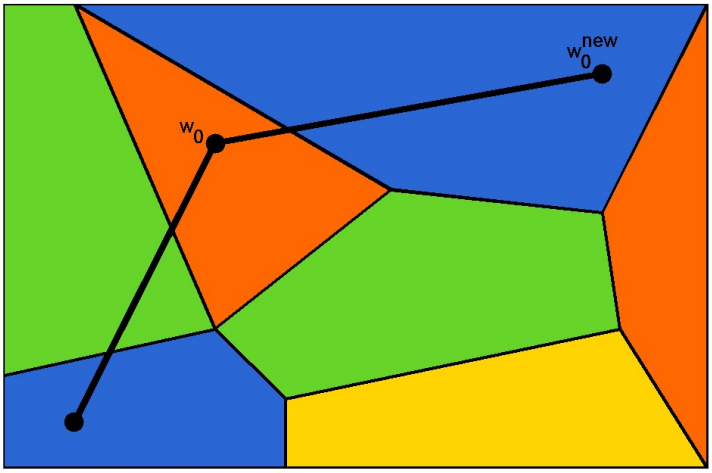
qpOASES User's Manual

Version 3.1 (January 2015)



Hans Joachim Ferreau et al.[1,2]
ABB Corporate Research, Switzerland
support@qpOASES.org

[1]past and present qpOASES developers and contributors in alphabetical order: Eckhard Arnold, Alexander Buchner, Holger Diedam, Hans Joachim Ferreau, Boris Houska, Dennis Janka, Christian Kirches, Manuel Kudruss, Aude Perrin, Andreas Potschka, Milan Vukov, Thomas Wiese, Sebastian F. Walter, Leonard Wirsching

[2]qpOASES has been initially released and developed at KU Leuven within the Optimization in Engineering Center (OPTEC), while current development is mainly supported by researchers at the Interdisciplinary Center for Scientific Computing (IWR) at Heidelberg University.

Contents

Chapter 1

Introduction

1.1 Scope of the Software

Model predictive control (MPC) is an advanced control strategy which allows to determine inputs of a given process that optimise the forecasted process behaviour. These inputs, or control actions, are calculated repeatedly using a mathematical *process model* for the prediction. In doing so, the fast and reliable solution of convex *quadratic programming* problems in real-time becomes a crucial ingredient of most algorithms for both linear and nonlinear MPC. The success of linear MPC—where just one *quadratic program (QP)* needs to be solved at each sampling instant—can even be attributed to the fact that highly efficient and reliable methods for QP solution have existed for decades, and that their computation times are much smaller than the required sampling times in typical applications.
On the other hand, quadratic programs also arise as sub-problems in sequential quadratic programming (SQP) methods, which require not only one but several QPs be solved during the iteration. SQP methods can be used for solving general nonlinear programs (NLPs) and are also an established tool for solving nonlinear MPC problems.

qpOASES is an open-source implementation of the recently proposed online active set strategy (see [4] and [6]; the main idea has been published earlier for a different class of problems in [2]). It was inspired by important observations from the field of parametric quadratic programming and builds on the expectation that the optimal active set does not change much from one quadratic program to the next. It has several theoretical features that make it particularly suited for model predictive control applications. The software package qpOASES implements these ideas and also incorporates important modifications to make the algorithm numerically more robust [12, 7].
qpOASES solving QPs of the following form:

$$\begin{aligned} \min_{x} \quad & \tfrac{1}{2} x^T H x + x^T g(w_0) \\ \text{s.t.} \quad & \mathrm{lb}A(w_0) \leq Ax \leq \mathrm{ub}A(w_0)\,, \\ & \mathrm{lb}(w_0) \leq \ x \ \leq \mathrm{ub}(w_0)\,, \end{aligned}$$

where the Hessian matrix is symmetric and positive (semi-)definite and the gradient vector as well as the bound and constraint vectors depend affinely on the parameter w_0.

1.2 Scope of this Manual

This manual is organised as follows: first, the installation of the qpOASES software package is explained in Chapter 2. Afterwards, a concise description of its main functionality is given in Chapter 3, which should enable you to use qpOASES within a couple of minutes. Chapter 4 describes special variants for QPs with varying matrices, simply bounded QPs as well as QPs with semi-definite Hessian matrices. Advanced functionality like obtaining status information, using the concept of a so-called initialised homotopy or exploiting QP sparsity is discussed in Chapter 5. Various interfaces to third-party software are presented in Chapter 6. Finally, Chapter 7 (which is mainly intended for software developers) provides some insight into the internal software design of qpOASES, options for further tuning of the algorithm as well as information on performing unit tests.

1.3 Further Support

Further information and user support can be found on

`http://www.qpOASES.org/` , which re-directs to
`https://projects.coin-or.org/qpOASES/` .

If you have got questions, remarks or comments on qpOASES, it is strongly encouraged to report them by creating a new ticket at the qpOASES webpage. In case you do not want to disclose your feedback to the public, you may send an e-mail to

`support@qpOASES.org` .

Also bug reports, source code enhancements or success stories are most welcome! If you want to receive latest information on qpOASES or participate in public discussions on future developments of the code, you should subscribe to the qpOASES mailing list. See

http://list.coin-or.org/mailman/listinfo/qpoases/

for more details.

1.4 Citing qpOASES

If you use qpOASES within your scientific work, we strongly encourage you to cite at least one of the following publications:

- Reference to the **software**:

```
@ARTICLE{Ferreau2014,
 author = {H.J. Ferreau and C. Kirches and A. Potschka and H.G. Bock and M. Diehl},
 title = {{qpOASES}: A parametric active-set algorithm for quadratic programming},
 journal = {Mathematical Programming Computation},
 year = {2014},
 volume = {6},
 number = {4},
 pages = {327--363},
 keywords = {qpOASES, parametric quadratic programming, active set method,
             model predictive control}
}
```

- Reference to the **online active set strategy**:

```
@ARTICLE{Ferreau2008,
 author = {H.J. Ferreau and H.G. Bock and M. Diehl},
 title = {An online active set strategy to overcome the limitations of explicit MPC},
 journal = {International Journal of Robust and Nonlinear Control},
 year = {2008},
 volume = {18},
 number = {8},
 pages = {816--830},
 keywords = {model predictive control, parametric quadratic programming,
             online active set strategy}
}
```

- Reference to the **webpage**:

```
@MISC{qpOASES2014,
 author = {H.J. Ferreau and A. Potschka and C. Kirches},
 title = {{qpOASES} webpage},
 howpublished = {http://www.qpOASES.org/},
 year = {2007--2014},
 keywords = {qpOASES, model predictive control, parametric quadratic programming,
             online active set strategy}
}
```

Acknowledgements

The initial version of the software has been partly developed within the framework of the REGINS-PREDIMOT European project whose financial support is acknowledged.
Further development of the code has been supported by Research Council KUL: CoE EF/05/006 Optimization in Engineering Center (OPTEC) and the Research Foundation – Flanders (FWO) where the main author held a 4-years PhD fellowship. Their financial support and permission to work on this open-source software project is gratefully acknowledged.

Chapter 2

Installation

The software package qpOASES is written in an object-oriented manner in C++ and comes along with fully commented source code files. Besides some standard C libraries[1] *no further external software packages are required*. Optionally, the LAPACK and BLAS libraries can be linked for performing internal linear algebra operations.

2.1 Standard Installation

For installing qpOASES under LINUX, perform the following steps:

1. *Obtain the latest stable release* of qpOASES from

   https://projects.coin-or.org/qpOASES/ ,

   either by saving the zipped archive `qpOASES-3.1.0.tgz` on your local machine or by checking out the latest stable branch, e.g. by running

   ```
   svn co https://projects.coin-or.org/svn/qpOASES/stable/3.1
   ```

   from you shell.

2. If you obtained a zipped archive, *unpack the archive:*

   ```
   gunzip qpOASES-3.1.0.tgz
   tar xf qpOASES-3.1.0.tar
   ```

   A new directory `qpOASES-3.1.0` will be created. From now on we refer to (the full path of) this directory (or the one you used to check out the latest stable branch) by `<install-dir>`. It contains seven sub-folders, namely

   - `bin` (to contain compiled executables and libraries),
   - `doc` (this manual and a DOXYGEN configuration file),
   - `examples` (source code of example files for setting up your own QP problems),
   - `include` (qpOASES header files),

[1] `math.h`, `stdio.h`, `string.h` (as well as `sys/time.h`, `sys/stat.h` or `windows.h` for runtime measurements)

- `interfaces` (interfaces to third-party software, see Chapter 6),
- `src` (qpOASES source files),
- `testing` (basic unit testing, see Section 7.4).

3. qpOASES is distributed under the terms of the GNU Lesser General Public License v2.1, which you can find in the file `<install-dir>/LICENSE.txt` or Appendix A of this manual. *Please read this licence file carefully before you proceed with the installation*, as you implicitly agree with this licence by using qpOASES!

4. If you want to use qpOASES via the provided third-party interfaces only, you can skip the following steps and proceed as described in Chapter 6. Otherwise continue with the

   Compilation of the qpOASES *library* `libqpOASES.a` *and test examples*:

   ```
   cd <install-dir>
   make
   ```

   This library `libqpOASES.a` provides the complete functionality of the qpOASES software package. It can be used by, e.g., linking it against a main function from the `examples` folder.

   The `make` also compiles a couple of test examples; executables are stored within the directory `<install-dir>/bin`.

5. *Running a simple test example*:

   Among others, an executable called `example1` should have been created; run it in order to test your installation:

   ```
   cd <install-dir>/bin
   ./example1
   ```

   If it terminates after successfully solving two QP problems, qpOASES has been successfully installed!

6. *Optional, create source code documentation*[2]*:*

   ```
   cd <install-dir>/doc
   doxygen doxygen.config
   ```

   Afterwards, you can open the file `<install-dir>/doc/html/index.html` with your favorite browser in order to view qpOASES's source code documentation.

2.2 Customised Installation

Installation on Windows or Mac OS X

It is also possible to natively install qpOASES on a WINDOWS or MAC OS X machine as it does not require any LINUX-specific commands. Installation on different operating systems is facilitated by the following means:

[2] All source code files are commented in a way suitable for the documentation system DOXYGEN [13].

1. *Customised Makefiles*: When calling `make`, the file `<install-dir>/make.mk` is used to select compiler settings that are tailored to different operating systems. The following settings are provided:

   - `make_linux.mk`, the default choice, for compiling under LINUX,
   - `make_cygwin.mk` for compiling under WINDOWS using CYGWIN,
   - `make_windows.mk` for compiling under WINDOWS using MICROSOFT® Visual Studio,
   - `make_osx.mk` for compiling under MAC OS X.

   Uncomment your preferred choice and run `make`.

2. *Compiling with* CMAKE: The file `<install-dir>/CMakeLists.txt` configures compilation on LINUX or WINDOWS by means of CMAKE. We refer to the CMAKE documentation for further details.

Static vs. Dynamic Library

qpOASES can be compiled into either a static or a dynamic library to be linked against the executable at runtime. Both variants are configured in the respective Makefiles.

Linking LAPACK and BLAS

By default, qpOASES runs self-contained not relying on external libraries. However, it is possible to use pre-compiled versions of the LAPACK and BLAS libraries for performing internal linear algebra operations. For doing so, the `make_*.mk` corresponding to your operating system needs to be adapted:

1. set the variable `REPLACE_LINALG` to `0` and
2. specify the installation path of the LAPACK and BLAS libraries by assigning `LIB_LAPACK` and `LIB_BLAS`.

Known Issues

qpOASES has been tested on a number of different hardware platforms, operating systems and compilers. In some cases, certain particularities require additional adjustments of the code. qpOASES provides built-in work-arounds for the following situations:

- If compilation fails due to the fact that the `snprintf()` function is not supported, you can uncomment line 41 within `<install-dir>/include/qpOASES/Types.hpp` and try to compile again.
- If compilation fails because certain mathematical functions are not supported, you can uncomment line 52 within `<install-dir>/include/qpOASES/Types.hpp` to set the `__NO_FMATH__` compiler flag and try to compile again.

In case you encounter compilation issues not listed here, please report them by creating a new ticket at the qpOASES webpage. When doing so, make sure you provide at least the following basic information:

- exact version number of your qpOASES release,
- name and version of your operating system,
- name and version of the compiler you are using,
- compiler settings and build system that you use to build qpOASES,
- detailed error log of your compiler,
- actions you already tried fixing the issue yourself.

Chapter 3

Getting Started

This chapter explains to you within a few minutes how to solve a quadratic programming (QP) problem, or a whole sequence of them, by means of qpOASES. At the end a tutorial example is presented that might serve as a template for your own QPs.

3.1 Outline

Core of qpOASES is the QProblem class which is able to store, process and solve convex quadratic programs using the online active set strategy; it makes use of several auxiliary classes (see Chapter 7). Except for special situations, the QProblem class is intended to be the only *user interface* to qpOASES's functionality.

For solving a series of convex quadratic programs with fixed Hessian and constraint matrix, the following steps are necessary:

1. create an instance of the QProblem class,
2. initialise your QProblem object and solve the first QP (specified by its QP matrices and vectors),
3. solve each following QP by passing its vectors to your QProblem object.

Now, we will explain these three steps in more detail. Various variants and special cases are treated in later chapters for the ease of presentation.

3.2 Main Steps

Creating an Instance of the QProblem Class

Creating an QProblem object is done by means of the following constructor

```
QProblem( int nV, int nC );
```

which takes the number of variables nV and the number of constraints nC of the quadratic program sequence to be solved. At the moment it is not possible to solve QP sequences with varying problem dimensions.

Summary of the first step:
You can create an instance `example` of the `QProblem` class with the following command:
`QProblem example( nV,nC );`

Initialisation and Solution of First QP

The second step requires to initialise all internal data structures of the `QProblem` object and the solution of the first QP. Both can be accommodated with a single call to the following function:

```
returnValue init( const real_t* const H,
                  const real_t* const g,
                  const real_t* const A,
                  const real_t* const lb,
                  const real_t* const ub,
                  const real_t* const lbA,
                  const real_t* const ubA,
                  int&                nWSR,
                  real_t* const       cputime
                  );
```

which takes the positive (semi-)definite Hessian matrix $H \in \mathbb{R}^{\mathrm{nV} \times \mathrm{nV}}$, the gradient vector $g \in \mathbb{R}^{\mathrm{nV}}$, the constraint matrix $A \in \mathbb{R}^{\mathrm{nC} \times \mathrm{nV}}$ the lower and upper bound vectors $\mathrm{lb}, \mathrm{ub} \in \mathbb{R}^{\mathrm{nV}}$ and the lower and upper constraints' bound vectors $\mathrm{lb}A, \mathrm{ub}A \in \mathbb{R}^{\mathrm{nC}}$ of the initial quadratic program. Equality constraints are imposed be setting the corresponding entries of lower and upper (constraints') bounds vectors to the same value.
All these data must be stored in arrays of type `real_t` (matrices stored row-wise, i.e. C style, in an one-dimensional array) with appropriate dimensions. If there are, for example, no upper bounds in your QP formulation, you can pass a null pointer instead of vector lb[1].
All `init` functions make deep copies of all vector arguments, thus afterwards you have to free their memory yourself. The matrix arguments `H` and `A` are not deep copied, so they must not be changed between consecutive calls to `qpOASES`.
The function `init` initialises all internal data structures, e.g. matrix factorisations, and *solves* the first quadratic program using the initial homotopy idea of the online active set strategy. The integer argument `nWSR` specifies the maximum number of working set recalculations to be performed during the initial homotopy (on output it contains the number of working set recalculations actually performed!). If `cputime` is not the null pointer, it contains the *maximum allowed* CPU time in seconds for the whole initialisation (and the actually required one on output). See Section 5.6 for further details.

The function `init` returns a status code (of type `returnValue`) which indicates whether the initialisation was successful; possible values are:

- `SUCCESSFUL_RETURN`: initialisation successful (including solution of first QP),
- `RET_MAX_NWSR_REACHED`: initial QP could not be solved within the given number of working set recalculations,

[1] If your QP does not comprise constraints (apart from bounds), you should make use of a special variant for simply bounded QPs (cf. Chapter 4).

- RET_INIT_FAILED (or a more detailed error code): initialisation failed.

If init indicates a SUCCESSFUL_RETURN, several functions enable you to obtain information about the solution of the first QP. The most important ones are:

- `returnValue getPrimalSolution( real_t* const xOpt ) const`

  that writes the optimal primal solution vector (dimension: nV) into the array xOpt, which has to be allocated (and freed) by the user;

- `returnValue getDualSolution( real_t* const yOpt ) const`

  that writes the optimal dual solution vector[2] (dimension: $\mathrm{nV} + \mathrm{nC}$) into the array yOpt, which has to be allocated (and freed) by the user;

- `real_t getObjVal( ) const`

  that returns the optimal objective function value.

Summary of the second step:
Having created an QProblem object example, it can be initialised together with solving the first QP with the following command: `example.init( H,g,A,lb,ub,lbA,ubA,nWSR,cputime );`

Solution of the Following QPs

If not only a single quadratic program but a whole sequence of QPs shall be solved—as it is the usual situation for an MPC problem—the next QP can be solved using the function:

```
returnValue hotstart( const real_t* const g_new,
                      const real_t* const lb_new,
                      const real_t* const ub_new,
                      const real_t* const lbA_new,
                      const real_t* const ubA_new,
                      int&                nWSR,
                      real_t* const       cputime
                      );
```

The next QP is specified by passing its gradient vector g_new, its lower and upper bound vectors lb_new and ub_new as well as its lower and upper constraints' bound vectors lbA_new and ubA_new (QP matrices are assumed to be constant). It is solved by means of the online active set strategy using at most nWSR working set recalculations or at most cputime seconds of CPU time (if not null). On output nWSR and cputime contain the number of

[2] We use the following definition of the Lagrange function to define the dual multipliers:

$$Hx^{\mathrm{opt}} + g(w_0) - A^T y^{\mathrm{opt}} = 0 \quad \Longleftrightarrow \quad \mathtt{H \cdot x + g - y[0...nV\text{-}1] - A^T y[nV...nV+nC\text{-}1]} = 0$$

The dual solution vector contains exactly one entry per lower/upper bound as well as exactly one entry per lower/upper constraints' bound. Positive entries correspond to active lower (constraints') bounds, negative entries to active upper (constraints') bounds and a zero entry means that both corresponding (constraints') bounds are inactive.

working set recalculations that were actually performed and the actually required CPU time for solving the next QP, respectively.

Like most qpOASES functions, hotstart returns a status code; possible values are:

- SUCCESSFUL_RETURN: QP has been solved,
- RET_MAX_NWSR_REACHED: QP could not be solved within the given number of working set recalculations,
- RET_HOTSTART_FAILED (or a more detailed error code): QP solution failed.

Summary of the third step:
Having created and initialised a QProblem object example, the next QP can be solved as follows: `example.hotstart( g_new,lb_new,ub_new,lbA_new,ubA_new,nWSR,cputime );`

3.3 A Tutorial Example

A complete example for solving two very simple quadratic programs using qpOASES is given in the file <install-dir>/examples/example1.cpp:

```
#include <qpOASES.hpp>

int main( )
{
    USING_NAMESPACE_QPOASES

    /* Setup data of first QP. */
    real_t H[2*2] = { 1.0, 0.0, 0.0, 0.5 };
    real_t A[1*2] = { 1.0, 1.0 };
    real_t g[2] = { 1.5, 1.0 };
    real_t lb[2] = { 0.5, -2.0 };
    real_t ub[2] = { 5.0, 2.0 };
    real_t lbA[1] = { -1.0 };
    real_t ubA[1] = { 2.0 };

    /* Setup data of second QP. */
    real_t g_new[2] = { 1.0, 1.5 };
    real_t lb_new[2] = { 0.0, -1.0 };
    real_t ub_new[2] = { 5.0, -0.5 };
    real_t lbA_new[1] = { -2.0 };
    real_t ubA_new[1] = { 1.0 };

    /* Setting up QProblem object. */
    QProblem example( 2,1 );

    /* Solve first QP. */
    int nWSR = 10;
    example.init( H,g,A,lb,ub,lbA,ubA, nWSR );
```

```
    /* Solve second QP. */
    nWSR = 10;
    example.hotstart( g_new,lb_new,ub_new,lbA_new,ubA_new, nWSR );

    /* Get and print solution of second QP. */
    real_t xOpt[2];
    example.getPrimalSolution( xOpt );
    printf( "\n xOpt = [ %e, %e ];  objVal = %e\n\n",
                          xOpt[0],xOpt[1],example.getObjVal() );

    return 0;
}
```

In order to access the functionality of the qpOASES software package via the QProblem class, the header file qpOASES.hpp is included.
The main function starts with defining the data of two very small-scale QPs. Afterwards, a QProblem object is created which is then initialised together with solving the first QP. Finally, the hotstart function is used to solve the second QP.
You might wonder about the command using namespace qpOASES; at the very top of the main function. It is used because all classes, global functions and variables of the qpOASES software package are *collected in a common namespace* that is called qpOASES, too.

3.4 Setting Up Your Own Example

The easiest way for setting up your own example, say yourexample, is to use an existing one as a template. In doing so, perform the following steps:

1. *Copy the existing example:*

   ```
   cd <install-dir>/examples
   cp example1.cpp yourexample.cpp
   ```

2. *Edit the examples* Makefile:

   Open the file <install-dir>/examples/Makefile and add a new target

   ```
   yourexample:  yourexample.o
       @${ECHO} "Creating" $@
       @${CPP} ${DEF_TARGET} ${CPPFLAGS} $< ${QPOASES_LINK} ${LINK_LIBRARIES}
   ```

   (Do not forget to add its name to the all target.)

3. *Implement your own example:*

   Modify your file <install-dir>/examples/yourexample.cpp and run make. An executable called <install-dir>/bin/yourexample should be at your service.

Chapter 4

Solution Variants for Special QP Types

qpOASES is a structure-exploiting active-set QP solver. This chapter details how to most efficiently solve your QPs with qpOASES by choosing a solution variant that matches best your specific problem type. For this purpose, three different QProblem-like classes with overloaded constructors are available.

4.1 Solving QPs in Standard Form

Usually qpOASES expects QPs to be formulated in the following *standard form*:

$$\begin{aligned} \min_{x} \quad & \tfrac{1}{2} x^T H x + x^T g(w_0) \\ \text{s.t.} \quad & \mathrm{lb}A(w_0) \leq Ax \leq \mathrm{ub}A(w_0)\,, \\ & \mathrm{lb}(w_0) \leq \ x \ \leq \mathrm{ub}(w_0)\,, \end{aligned}$$

with a positive (semi-)definite Hessian matrix H. If your QP is given in exactly this form, you should simply make use of the standard QProblem class as described in Chapter 3. Otherwise, solving your QP is also possible or can be done more efficiently if

- also your QP matrices H and/or A are varying from one QP to the next by using the SQProblem class (see Section 4.2);
- your QP formulation does *not* comprise constraints involving a constraint matrix A by using the QProblemB class (see Section 4.3);
- your Hessian matrix H is not positive definite but only positive semi-definite by using a dedicated constructor (see Section 4.4);
- your Hessian matrix H is zero, i.e. your QP is actually a linear program, by using a dedicated constructor (see Section 4.5);
- your Hessian matrix H happens to be the identity matrix by using a dedicated constructor (see also Section 4.5).

4.2 Solving QPs with Varying Matrices

Although the online active set strategy was originally designed for QP sequences with fixed Hessian and constraint matrices, it can be extended to the case where also these matrices vary from QP to the next (see [5] for a mathematical description of this idea).
In order to use this extension, two modifications are necessary:

1. Create an instance of the `SQProblem` class (instead of one of type `QProblem`) by using the constructor of the `SQProblem` class and a suitable `init` function. Both take *exactly the same* arguments as those of the `QProblem` class.

2. Call the modified function

   ```
   returnValue hotstart( const real_t* const H_new,
                         const real_t* const g_new,
                         const real_t* const A_new,
                         const real_t* const lb_new,
                         const real_t* const ub_new,
                         const real_t* const lbA_new,
                         const real_t* const ubA_new,
                         int&                nWSR,
                         real_t* const       cputime
                         );
   ```

   for transition from one QP to the next; it also takes the new Hessian `H_new` as well as the new constraint matrix `A_new` as arguments.

A complete example for using the `SQProblem` class can be found within the file `<install-dir>/examples/example1a.cpp`.

4.3 Solving Simply Bounded QPs

We call a quadratic program "simply bounded" whenever it does not comprise constraints but only bounds:

$$\begin{aligned} \min_{x} \quad & \tfrac{1}{2} x^T H x + x^T g(w_0) \\ \text{s.t.} \quad & \mathrm{lb}(w_0) \leq \ x \ \leq \mathrm{ub}(w_0) \,. \end{aligned}$$

This special form can be exploited within the solution algorithm for speeding up the computation, typically by a factor of three to five. Therefore, the qpOASES software package implements the special class `QProblemB` for solving simply bounded QPs.

In order to make use of this feature do the following:

1. You have to create a `QProblemB` object using the following constructor

   ```
   QProblemB( int nV );
   ```

2. Afterwards you can initialise the `QProblemB` object together with solving the first simply bounded QP by calling, for example,

```
returnValue init( const real_t* const H,
                  const real_t* const g,
                  const real_t* const lb,
                  const real_t* const ub,
                  int&                nWSR,
                  real_t* const       cputime
                  );
```

The only difference from the QProblem class is the fact that the arguments specifying the constraints—i.e. A, lbA, ubA, and nC—are missing.

3. For solving the next problem within your QP sequence, the following variant of the hotstart function is available:

```
returnValue hotstart( const real_t* const g_new,
                      const real_t* const lb_new,
                      const real_t* const ub_new,
                      int&                nWSR,
                      real_t* const       cputime
                      );
```

Again, it takes exactly the same arguments as the corresponding QProblem member function except for the two arguments lbA_new, ubA_new.

A complete example for using the QProblemB class can be found within the file <install-dir>/examples/example1b.cpp.

4.4 Solving QPs with Positive Semi-Definite Hessian Matrix

qpOASES provides two different strategies to deal with semi-definite QPs. All mentioned options to enable and to adjust these strategies are described in more detail in Section 5.2.

Automatic Regularisation Procedure

The first one is a regularisation procedure that is computationally cheap and works well for many problems. This procedure first adds a small multiple of the identity matrix[1] to the Hessian and solves the corresponding regularised QP. Afterwards, a few post-iterations[2] are performed that improve solution accuracy significantly over a plain regularisation at virtually now extra computational cost. If your QP involves a Hessian matrix that is only positive semi-definite, this regularisation scheme *is used automatically*, i.e. without any change in the constructor or other function calls, whenever the option enableRegularisation is set to BT_TRUE.

Although semi-definiteness can be easily detected, this causes a certain computational overhead[3] that can be avoided by a dedicated constructor call, e.g.:

[1] given by the option epsRegularisation
[2] given by the option numRegularisationSteps
[3] an additional Cholesky decomposition

```
QProblem( int nV, int nC, HessianType hessianType );
```

Therein, hessianType can take one of the following values:

- HST_POSDEF: Hessian matrix is positive definite,
- HST_SEMIDEF: Hessian matrix is positive semi-definite,
- HST_INDEF: Hessian matrix is indefinite,
- HST_ZERO: Hessian matrix is zero matrix (see next section),
- HST_IDENTITY: Hessian matrix is identity matrix (see next section).

If hessianType is set to HST_SEMIDEF or HST_ZERO, the built-in regularisation scheme is switched on at no additional computational cost. Corresponding overloaded constructors also exist for the SQProblem and QProblemB class, respectively.

In case the build-in Cholesky decomposition is used and needs to abort due to a negative diagonal element, the magnitude of this diagonal element is returned and used to automatically increase the option epsRegularisation. Afterwards, the automatic regularisation procedure is re-started once applying this increased regularisation parameter.

Nonzero Curvature Tests and Flipping Bounds

A second strategy to deal with semi-definite QPs is the use of nonzero curvature tests as described in [2]. The main idea is to check upon removal of an active constraint or bound whether the projected Hessian matrix will loose full rank. If so, another constraint or bound is immediately added to the active set to ensure full rank of the projected Hessian matrix. Nonzero curvature tests can be enabled by setting the option enableNZCTests to BT_TRUE[4].

Nonzero curvature tests can be combined with the use of flipping bounds as proposed in [12]: In case removal of an active constraint or bound causes the smallest eigenvalue of the projected Hessian matrix to drop below a small positive threshold, the constraint or bound remains active but the intermediate QP data is changed such that it is active at its opposite limit (e.g. an active upper bound will become an active lower bound). This also prevents the Cholesky decomposition from becoming ill-conditioned in case the Hessian matrix is positive definite with very small positive eigenvalues. Flipping bounds can be enabled by setting the option enableFlippingBounds to BT_TRUE. An option epsFlipping can be used to adjust the lower threshold allowed for the smallest positive eigenvalue of the projected Hessian matrix.

The flipping bound strategy requires the initial projected Hessian matrix to be positive definite. The option initialStatusBounds provides an easy way to ensure this by initially fixing all bound constraints to their respective lower or upper limit (that way the projected Hessian matrix has zero dimension). Alternatively, an initial guess for the active set as described in Section 5.5 might be used to ensure positive definiteness of the projected Hessian matrix.

[4] Note that this feature is only available when using the classes QProblem or SQProblem.

Unbounded QPs

In case the Hessian matrix is only positive semi-definite, the objective function might be unbounded from below (i.e. the QP does not have an optimal solution). This can be excluded by imposing lower and upper bounds on all QP variables. If you are uncertain whether your QP may be unbounded or not, it is strongly advised *not* to use the automatic regularisation procedure, but to enable both the flipping bound and the far bound strategy instead.

4.5 Solving QPs with Trivial Hessian Matrix

Whenever a Hessian matrix is passed to qpOASES, i.e. when calling a `init` function or performing a `hotstart` while using the `SQProblem` class, it is internally checked whether the Hessian is trivial. It is considered trivial if and only if it is the zero or identity matrix, corresponding to `HST_ZERO` or `HST_IDENTITY` as mentioned in the previous section. If the Hessian is trivial, several simplifications of the internal linear algebra operations apply, cutting computational load by about a factor of two.

If your Hessian is trivial, you might explicitly provide this information to qpOASES via a dedicated constructor call, e.g.,

```
QProblem( int nV, int nC, HessianType hessianType );
```

(corresponding overloaded constructors also exist for the `SQProblem` and `QProblemB` class, respectively). If you set `hessianType` to `HST_ZERO` or `HST_IDENTITY`, no internal memory for storing the Hessian matrix is allocated. Moreover, when doing so you are allowed to pass a null pointer as argument within all function calls involving the Hessian matrix, e.g.,

```
// assumes that a QProblem object "qp" exists
qp.init( 0,g,A,lb,ub,lbA,ubA,nWSR,cputime );
```

A null pointer is then interpreted as zero or identity matrix, respectively. Whenever you pass a non-null argument, a full Hessian matrix is expected and its type is automatically determined internally.

Solving Linear Programming (LP) Problems

Both strategies mentioned in Section 4.4 in principle also allow to solve linear programming (LP) problems by means of qpOASES. However, qpOASES *is not a dedicated (parametric) LP solver*, thus using it for solving LPs might be (highly) inefficient due to the dense linear algebra or might even fail in certain circumstances. Therefore, this additional feature should be only used for small-scale LPs (comprising not more than, say, hundreds of variables) and in situations where computational time is not the main concern.

A complete example for solving two small-scale LPs with qpOASES can be found within the file `<install-dir>/examples/exampleLP.cpp`.

Solving QPs whose Hessian is the Identity Matrix

Via a coordinate transformation, every strictly convex QP can be transformed into an equivalent one whose Hessian is the identity matrix. Also ℓ_2-norm minimisation problems naturally pose QPs whose Hessian is the identity matrix. Thus, it is possible to provide such a QP sequence to qpOASES by specifying the Hessian type to be `HST_IDENTITY` within the above-mentioned constructor call; all other function calls remain unaltered.

4.6 Solving Non-Convex QPs

Due to the flipping bounds strategy, qpOASES can also be used to find critical points of non-convex QP problems. A more detailed description will follow.

Chapter 5

Advanced Functionality

5.1 Obtaining Status Information

There are many functions for obtaining status information on the current iterate. Firstly, you can obtain the primal and dual iterate as well as the corresponding objective function value by using, respectively:

- `returnValue getPrimalSolution( real_t* const xOpt ) const`,
- `returnValue getDualSolution( real_t* const yOpt ) const`,
- `real_t getObjVal( ) const`.

If you wonder why these are the same functions as for obtaining the optimal solution after a QP has been solved (cf. Section 3.2), you should recall that qpOASES uses a homotopy for solving the current QP that produces a sequence of iterates that are *optimal for intermediate QPs* along the homotopy path.
The first two functions expect an allocated `real_t` array and store the optimal solution vector if and only if the (intermediate) QP has been solved; otherwise the error code `RET_QP_NOT_SOLVED` is returned. The function `getObjVal( )` calculates and returns the optimal objective function value or returns `INFTY` if the (intermediate) QP has not been solved.

Secondly, you can ask for the total number of variables and constraints and for the cardinality of certain subsets (at current iterate!) of them:

- `int getNV( ) const`: returns number of variables,
- `int getNFR( ) const`: returns number of free variables,
- `int getNFX( ) const`: returns number of fixed variables,
- `int getNC( ) const`: returns number of constraints,
- `int getNEC( ) const`: returns number of (implicitly defined) equality constraints,
- `int getNAC( ) const`: returns number of active constraints,

- `int getNIAC( ) const` : returns number of inactive constraints.

Moreover,

- `int getNZ( ) const` : returns dimension of the null space of active constraints.

Finally, you can ask for the overall status of the QP (object):

- `BooleanType isInitialised( ) const` : returns `BT_TRUE` if and only if the QP object has been initialised,
- `BooleanType isSolved( ) const` : returns `BT_TRUE` if and only if QP has been solved,
- `BooleanType isInfeasible( ) const` : returns `BT_TRUE` if and only if QP was found to be infeasible,
- `BooleanType isUnbounded( ) const` : returns `BT_TRUE` if and only if QP was found to be unbounded (from below).

5.2 Options for Solving QPs

The way qpOASES solves QPs can be adjusted in several ways by means of the class `Options`. It comprises the following members whose values can be set by the user:

Name:	*Possible values:*	*Description:*

`printLevel`	`PL_NONE`
	`PL_LOW`
	`PL_MEDIUM`
	`PL_HIGH`
	`PL_TABULAR`
	`PL_DEBUG_ITER`
		Defines the amount of text output during QP solution, see Section 5.8.

`enableRamping`	`BT_TRUE` `BT_FALSE`	Enables or disables ramping, an idea to avoid ties when determining the step length [12].
`enableFarBounds`	`BT_TRUE` `BT_FALSE`	Enables or disables the use of far bounds, an idea to reliably detect unboundedness [12].
`enableFlippingBounds`	`BT_TRUE` `BT_FALSE`	Enables or disables the use of flipping bounds as described in Section 4.4.
`enableRegularisation`	`BT_TRUE` `BT_FALSE`	Enables or disables the Hessian regularisation scheme as described in Section 4.4.
`enableFullLITests`[1]	`BT_TRUE` `BT_FALSE`	Enables or disables a condition-hardened, but more expensive test for linear independence.
`enableNZCTests`[1]	`BT_TRUE` `BT_FALSE`	Enables or disables nonzero curvature tests as described in Section 4.4.

[1] Note that this option is *not* available when using the class `QProblemB`.

enableDriftCorrection	int (≥ 0)	Specifies the frequency of drift corrections [12]: 0 turns them off, 1 uses them at each iteration etc.
enableCholeskyRefactorisation	int (≥ 0)	Specifies the frequency of full refactorisations of the projected Hessian: 0 turns them off, 1 uses them at each iteration etc.
enableEqualities	BT_TRUE BT_FALSE	Specifies whether equalities shall be always treated as active constraints.
terminationTolerance	real_t (> 0)	Relative termination tolerance to stop homotopy.
boundTolerance	real_t (> 0)	If upper and lower limits differ less than this tolerance, they are regarded equal, i.e. as equality constraint.
boundRelaxation	real_t (> 0)	Initial relaxation of bounds to start homotopy and initial value for far bounds.
epsNum	real_t	Numerator tolerance for ratio tests.
epsDen	real_t	Denominator tolerance for ratio tests.
maxPrimalJump	real_t (> 0)	Maximum jump in primal variables allowed in nonzero curvature tests.
maxDualJump	real_t (> 0)	Maximum jump in dual variables allowed in linear independence tests.
initialRamping	real_t (> 0)	Start value for ramping strategy.
finalRamping	real_t (> 0)	Final value for ramping strategy.
initialFarBounds	real_t (> 0)	Initial size of far bounds.
growFarBounds	real_t (> 1)	Factor to grow far bounds.
initialStatusBounds	ST_INACTIVE ST_LOWER ST_UPPER	Initial status of bounds at first iteration: all inactive or all active at their lower or upper limits, respectively.
epsFlipping	real_t (> 0)	Tolerance of squared entry on Cholesky diagonal which triggers flipping bound.
numRegularisationSteps	int (≥ 0)	Maximum number of successive regularisation steps.
epsRegularisation	real_t (> 0)	Scaling factor of identity matrix used for Hessian regularisation.
numRefinementSteps	int (≥ 0)	Maximum number of iterative refinement steps.
epsIterRef	real_t (> 0)	Early termination tolerance for iterative refinement [12].
epsLITests[2]	real_t (> 0)	Tolerance for linear independence tests.
epsNZCTests[2]	real_t (> 0)	Tolerance for nonzero curvature tests.

[2]Note that this option is *not* available when using the class QProblemB.

If the user does not specify any options, default values are used. For changing these default values, the following steps are required:

1. Create an Options object and modify any of the above mentioned options as follows

```
Options myOptions;
myOptions.<optionName> = <optionValue>;
```

2. Pass your options to the QP object:

```
// assumes that a QP object "qp" exists
qp.setOptions( myOptions );
```

In order to facilitate the choice of reasonable values for all these options, the Options class offers a couple of pre-defined configurations:

- setToDefault( ); assigns default values to all options,
- setToReliable( ); chooses values that ensure maximum reliability of the QP solution (usually at the expense of a slower execution),
- setToMPC( ); chooses values that ensure maximum computational speed that might lead to a failure of the algorithm in certain cases.

Thus, a complete example could look like:

```
Options myOptions;
myOptions.setToMPC( );
myOptions.printLevel = PL_LOW;
qp.setOptions( myOptions );
```

Note that changing options will take effect immediately after passing them.

5.3 Exploiting Sparsity in Hessian and Constraints Matrix

qpOASES has been developed for small- to medium scale QPs resulting from MPC formulations after the differential states have been eliminated. These QPs usually feature a fully dense Hessian matrix and a lower triangular constraint matrix. Consequently the whole internal linear algebra—including the matrix factorisations—is implemented dense. For enhancing qpOASES's applicability to general QPs, a minimalistic Matrix base class has been introduced. This framework also supports sparse QP matrices and allows one to use special linear algebra routines for symmetric matrices.

For passing sparse QP matrices, overloaded variants of all init and hotstart routines exists. These variants do not read the Hessian and constraints matrix from real_t arrays but rather expect them in form of derived classes of the minimalistic Matrix base class, for example:

```
returnValue init( SymmetricMatrix*    H,
                  const real_t* const g,
                  Matrix*             A,
                  const real_t* const lb,
                  const real_t* const ub,
                  const real_t* const lbA,
                  const real_t* const ubA,
                  int&                nWSR,
                  real_t* const       cputime
                  );
```

General dense matrices are stored within instances of the class `DenseMatrix`, general sparse matrices within ones of the class `SparseMatrix`. For symmetric matrices the classes `SymDenseMat` and `SymSparseMat` are provided, respectively. Sparse matrices are stored in column compressed storage format. We refer to the DOXYGEN source code documentation for further details.

A complete tutorial example illustrating the use of sparse QP matrices can be found within the file <install-dir>/examples/qrecipe.cpp.

5.4 Specifying a Function for Evaluating the Constraints

Another possibility to speed-up QP solution in case of many constraints is available whenever the calculation of the matrix-product of the constraint matrix `A` with the current primal iterate `x` can be simplified. In that case, the user can provide a dedicated function that can evaluate the product of any constraint at a given primal iterate. Once such a function is specified and passed to an QP object, qpOASES will use this user-provided function for calculating the constraint products instead of doing a standard (but possibly naive) matrix-vector multiplication.

For using this functionality, you have to perform the following steps:

1. Derive a customized class from the abstract base class `ConstraintProduct` as declared within <install-dir>/include/ConstraintProduct.hpp. Within this class, you have to implement the function operator which has the following form:

   ```
   virtual int operator()( int constrIndex,
                           const real_t* const x,
                           real_t* const constrValue
                           ) const;
   ```

   It takes the index of the constraint to be evaluated (between `0` and `nC`) and an array containing the current primal iterate (of size `nV`) as input arguments and writes the corresponding product into `constrValue`. The function operator needs to return `0` on success and might return an error code otherwise.

2. Make this derived class available within your example, instantiate an object of this class and pass it to the QP object by calling

```
// assumes that a QP object "qp" exists
MyConstraintProduct myCP( );
qp.setConstraintProduct( &myCP );
```

A full tutorial example illustrating this feature of qpOASES can be found within the file <install-dir>/examples/example4.cpp.

5.5 Initialised Homotopy

For solving a QP, qpOASES always starts at the optimal solution of the previous QP and performs a homotopy to the optimal solution of the QP to be solved. At the very beginning of a sequence (when init is called) an auxiliary QP is constructed internally whose optimal solution is known. This optimal solution serves as a starting point for the homotopy to the optimal solution of the (actual) initial QP. By default, this auxiliary QP has the origin as solution and its active set is empty (or comprising implicitly fixed variables and equality constraints only).

The notion *initialised homotopy* refers to the possibility to incorporate an initial guess for the optimal solution or the active set at the solution into the construction of the auxiliary QP. This is done by calling a special variant of the init function:

```
returnValue init( const real_t* const      H,
                  const real_t* const      g,
                  const real_t* const      A,
                  const real_t* const      lb,
                  const real_t* const      ub,
                  const real_t* const      lbA,
                  const real_t* const      ubA,
                  int&                     nWSR,
                  real_t* const            cputime,
                  const real_t* const      xOpt,
                  const real_t* const      yOpt,
                  const Bounds* const      guessedBounds,
                  const Constraints* const guessedConstraints
                  );
```

Besides the arguments of the usual init function, it (optionally) takes guesses for the primal solution vector xOpt, the dual solution vector yOpt or the status (active/inactive) of bounds and constraints at the solution (see below). Null pointers can be passed for all of these arguments. The construction of the auxiliary QP now depends on the arguments passed (for convenience we summarise guessedBounds and guessedConstraints to guess which is null if and only if both parts are null) as follows:

1. xOpt == 0, yOpt == 0, guess == 0: start at primal/dual origin with empty active set (usual auxiliary QP setup);
2. xOpt != 0, yOpt == 0, guess == 0: start at primal/dual origin and determine active set by "clipping"[1];

[1] i.e. add all bounds and constraints to active set that are violated for given primal solution vector

3. `xOpt == 0, yOpt != 0, guess == 0`: start with primal variables equal to zero, dual variables equal to given vector and determine active set from signs of dual variables;

4. `xOpt == 0, yOpt == 0, guess != 0`: start at primal/dual origin and with given active set;

5. `xOpt != 0, yOpt != 0, guess == 0`: start with given vectors for primal and dual variables and determine active set from signs of dual variables;

6. `xOpt != 0, yOpt == 0, guess != 0`: start with primal variables equal to given vector, dual variables equal to zero and with given active set;

7. `xOpt != 0, yOpt != 0, guess != 0`: start with given vectors for primal and dual variables and with given active set (assume them to be consistent!).

The remaining eighth combination is not allowed for consistency reasons.

Besides initialising the homotopy at start-up of the QP sequence, it is also possible to incorporate an initial guess for the active set when calling the `hotstart` function:

```
returnValue hotstart( const real_t* const      g_new,
                      const real_t* const      lb_new,
                      const real_t* const      ub_new,
                      const real_t* const      lbA_new,
                      const real_t* const      ubA_new,
                      int&                     nWSR,
                      real_t* const            cputime,
                      const Bounds* const      guessedBounds,
                      const Constraints* const guessedConstraints
                      );
```

In this case only the active set can be specified, primal and dual solution vectors are always taken from the previous QP solution. This `hotstart` variant updates the active set according to the user's guess and performs a usual homotopy afterwards.

Specifying an Initial Guess for the Active Set

For specifying an initial guess for the active set, you have to setup a `Bounds` and/or `Constraints` object. This can either be done from scratch or by modifying an existing one. For the first variant you might use the following code fragment:

```
// assumes that a QP object "qp" exists
int nV = qp.getNV( );
int nC = qp.getNC( );

Bounds guessedBounds( nV );
guessedBounds.setupAllLower( );

Constraints guessedConstraints( nC );
guessedConstraints.setupAllInactive( );
```

First, a Bounds object comprising a working set of nV bounds is constructed and afterwards all bounds are set to be active at their lower limit. Second, a Constraints object is constructed analogously and all constraints are set to be inactive. For a Bounds object you can call one of the following functions:

- returnValue setupAllFree( ): all variables are free, i.e. bounds are inactive,
- returnValue setupAllLower( ): all variables are fixed at their lower limits,
- returnValue setupAllUpper( ): all variables are fixed at their upper limits.

For a Constraints object you can call one of the following functions:

- returnValue setupAllInactive( ): all constraints are inactive,
- returnValue setupAllLower( ): all constraints are active at their lower limits,
- returnValue setupAllUpper( ): all constraints are active at their upper limits.

Moreover, you might setup the status of each bound/constraint one by one by calling:

- returnValue setupBound(int number, SubjectToStatus status) or
- returnValue setupConstraint(int number, SubjectToStatus status),

respectively, where number specifies the number of the respective bound/constraint (starting at zero!) and status is one of the following types:

- ST_INACTIVE: bound/constraint is inactive,
- ST_LOWER: bound/constraint is active at its lower limit,
- ST_UPPER: bound/constraint is active at its upper limit.

Please note that you can call *either* exactly one setupAll* variant *or* exactly one of setupBound/setupConstraint for each single bound/constraint!

Instead of setting up a Bounds/Constraints object from scratch, you might want to *modify an existing one*. For achieving this, you will most commonly first obtain a copy of the active set of the current QP by calling:

```
// assumes that a QP object "qp" exists
Bounds guessedBounds;
qp.getBounds( guessedBounds );

Constraints guessedConstraints;
qp.getConstraints( guessedConstraints );
```

Afterwards you might use one of the following functions to manipulate a Bounds object:

- returnValue moveFixedToFree(int number): moves the number-th bound from the working set of fixed variables to that of free ones,

- `returnValue moveFreeToFixed( int number, SubjectToStatus status )`: moves the `number`-th bound from the working set of free variables to that of fixed ones (where `status` must be either `ST_LOWER` or `ST_UPPER`).

For a `Constraints` object you can call one of the following functions:

- `returnValue moveActiveToInactive( int number )`: moves the `number`-th constraint from the working set of active constraints to that of inactive ones,
- `returnValue moveInactiveToActive( int number, SubjectToStatus status )`: moves the `number`-th constraint from the working set of inactive constraints to that of active ones (where `status` must be either `ST_LOWER` or `ST_UPPER`).

Moreover, in the model predictive control context it is very common that the active set is *shifted* between two consecutive sampling instants. Therefore, for both `Bounds` and `Constraints` you can also call one of the following functions:

- `returnValue shift( int offset )`: shifts forward the working set of bounds/constraints by a given offset (which has to be an integer divisor of the total number of bounds/constraints), i.e. the status information of the first `offset` bounds/constraints is thrown away and the one of the last `offset` ones is duplicated;
- `returnValue rotate( int offset )`: rotates forward the working set of bounds/constraints by a given offset.

We refer to the DOXYGEN documentation (cf. installation step six described in Chapter 2) for more details.

5.6 Specifying a CPU Time Limit for QP Solution

For all `init` and `hotstart` function calls the input argument `nWSR` is mandatory. Additionally, it is possible to specify a maximum amount of CPU time to be spent on the respective QP solution. For doing so, a non-null pointer to a `real_t` containing the maximum allowed CPU time in seconds needs to be specified. If both, a maximum number of working set recalculations `nWSR` and a maximum allowed CPU time `cputime` is given, the solution procedure stops as soon as *one of these limits* is reached, whatever may occur first.

The CPU time limitation is based on a *heuristic* that estimates the required CPU time for the next working set change; if there is not enough time left, the solution procedure stops. This heuristic is based on the CPU time measurements of the previous working set changes, thus the actual total CPU time might be slightly higher that the allowed one due to time measurement inaccuracies. However, it is guaranteed that *at most one* working set change too much is performed.

Note that the CPU time limit only can take effect if a system clock is available via the global `getCPUtime` function (implemented within the file `src/Utils.cpp`).

5.7 Providing a Pre-computed Cholesky Factor

If the option initialStatusBounds is set to ST_INACTIVE and no initial guess of the optimal active set is provided, qpOASES starts the QP solution procedure with computing a Cholesky decomposition of the (possibly regularised) Hessian matrix. This can make the initial call to the init function significantly more expensive than that to subsequent hotstart calls. In case the Hessian matrix is fixed, its Cholesky factor may be pre-computed offline and passed to qpOASES in order to reduce online computational load. For doing so, a special variant of the init function is provided:

```
returnValue init( const real_t* const      H,
                  const real_t* const      g,
                  const real_t* const      A,
                  const real_t* const      lb,
                  const real_t* const      ub,
                  const real_t* const      lbA,
                  const real_t* const      ubA,
                  int&                     nWSR,
                  real_t* const            cputime,
                  const real_t* const      xOpt,
                  const real_t* const      yOpt,
                  const Bounds* const      guessedBounds,
                  const Constraints* const guessedConstraints,
                  const real_t* const      _R
                  );
```

Passing null pointers for the arguments xOpt, yOpt, guessedBounds, guessedConstraints, the last argument _R may contain an upper-triangular Cholesky factor of the Hessian matrix H that satisfies

$$\_\mathrm{R}^T \cdot \_\mathrm{R} = \mathrm{H}\,.$$

The dense Cholesky factor _R must be stored in row-major format in an array of size nV*nV.

5.8 Further Useful Functionality

Reading Data From Files

Both the init and the hotstart functions are overloaded with variants that are able to read the required data directly from a plain ASCII file, e.g.:

- ```
  returnValue init( const char* const H_file,
                    const char* const g_file,
                    const char* const A_file,
                    const char* const lb_file,
                    const char* const ub_file,
                    const char* const lbA_file,
                    const char* const ubA_file,
                    int&              nWSR,
                    real_t* const     cputime
                    );
  ```
  ```

- `returnValue hotstart( const char* const g_file,`
  `const char* const lb_file,`
  `const char* const ub_file,`
  `const char* const lbA_file,`
  `const char* const ubA_file,`
  `int& nWSR,`
  `real_t* const cputime`
  `);`

Instead of a `real_t` array, they expect a string with the name of the ASCII file containing the respective data. Data files must be stored row-wise; all entries within one row should be space- or tabulator-separated.
These variants also exists for the case when an initial guess for the active set or an pre-computed Cholesky factor is provided (as described in Sections 5.5 and 5.7).

Output Settings

You can adjust the text output of qpOASES using the following functions:

- `PrintLevel getPrintLevel( ) const`,
- `void setPrintLevel( PrintLevel _printlevel )`.

The function `getPrintLevel` returns one of the following print levels:

- `PL_NONE`: no output at all,
- `PL_LOW`: print error messages only,
- `PL_MEDIUM`: print error messages, warnings, some info messages as well as a concise iteration summary (default value),
- `PL_HIGH`: print all messages that occur while iterating,
- `PL_TABULAR`: similar to `PL_MEDIUM` but displaying more detailed information in tabular form,
- `PL_DEBUG_ITER`: similar to `PL_TABULAR` but displaying even more detailed information in tabular form.

By means of the function `setPrintLevel` you can specify one of the above-mentioned print levels whenever desired.

Resetting a `QProblem` Object

Sometimes it can be useful to reset an existing `QProblem` object. This is particularly helpful if you want to restart while solving a QP sequence (e.g. after an internal error has occurred) without creating a new object. This feature is provided by the following function:

```
returnValue reset( );
```

It resets all internal data structures and matrix factorisations and thus leaves the QProblem object in exactly the same state as it would be after a constructor call. Therefore, you need to call an init function for solving the first QP after an execution of reset.

Printing QP Properties and Options

At any time you might print a concise list of properties of the QP object by calling:

```
returnValue printProperties( );
```

Besides other information, it displays number and type of bounds and constraints, respectively, the type of the Hessian matrix as well as the status of the QP object.

Moreover, a list of all options and their current values (see Section 5.2) can be printed as follows:

```
returnValue printOptions( );
```

Obtain Simple Status Flag

Both the init and the hotstart functions of an QP object return a specific returnValue indicating whether the current QP problem has been solved successfully or whether some kind of error has occurred. This returnValue can be converted into a less detailed status flag, which is more easy to interpret, by calling

```
int getSimpleStatus( returnValue returnvalue );
```

This function returns one of the following values:

- 0: QP was solved,
- 1: QP could not be solved within the given number of iterations,
- -1: QP could not be solved due to an internal error,
- -2: QP is infeasible and thus could not be solved,
- -3: QP is unbounded and thus could not be solved.

Retrieve QP Problem Counter

qpOASES internally counts the number of QP problems solved with a given (S)QProblem(B) object. This number can be retrieved by calling:

```
unsigned int getCount( );
```

Note that this counter is *not* reset when calling the reset function mentioned above. Instead, the counter may be reset independently by calling:

```
returnValue resetCounter( );
```

5.9 Add-Ons for qpOASES

There exists a couple of add-ons to qpOASES's core functionality that implement features that go beyond solving QP problems[2]. This subsection describes these features in more detail.

5.9.1 Solution Analysis

For a posteriori analysis of a QP solution the `SolutionAnalysis` class is provided as an add-on to qpOASES. Currently it implements the following two functions:

- Determination of the maximum violation of the KKT optimality conditions:

```
real_t getKktViolation( QProblem* const qp,
                        real_t* const maxStat,
                        real_t* const maxFeas,
                        real_t* const maxCmpl
                        ) const;
```

  This function takes a pointer to a `QProblem` object which is assumed to have readily solved an (intermediate) QP and returns the maximum violation of the KKT optimality conditions. The arguments `maxStat`, `maxFeas` and `maxCmpl` or optional and may contain the maximum violation of the stationarity, feasibility and complementary conditions, respectively.

- Computation of the variance-covariance matrix of the QP output for uncertain inputs:

```
returnValue getVarianceCovariance( QProblem* const qp,
                                   const real_t* const g_b_bA_VAR,
                                   real_t* const Primal_Dual_VAR
                                   ) const;
```

  It also takes a `QProblem` object which is assumed to have readily solved an (intermediate) QP as well as the variance-covariance of the gradient, the bounds and the constraints' bounds, respectively (matrix dimension: $2\mathtt{nV}+\mathtt{nC} * 2\mathtt{nV}+\mathtt{nC}$). The variance-covariance matrix of the primal and dual variables is written into the argument `Primal_Dual_VAR` (matrix dimension: $2\mathtt{nV}+\mathtt{nC} * 2\mathtt{nV}+\mathtt{nC}$), which needs to be allocated by the user.

For using the `SolutionAnalysis` class you need to include its header `SolutionAnalysis.hpp` into your source file, a complete example can be found in the file `<install-dir>/examples/example2.cpp`.

[2] Previous versions of qpOASES were building two libraries `libqpOASES.a` (containing only the core functionality) and `libqpOASESextras.a` (also including the add-ons). This distinction has been skipped for streamlining the build process. However, if object code size is at a premium, these add-ons can be left away by excluding the files `OQPinterface.cpp` and `SolutionAnalysis.cpp` from the build process.

5.9.2 Solving Test Problems from the Online QP Benchmark Collection

A second qpOASES add-on is intended to facilitate the solution of test problems from the Online QP Benchmark Collection [1]. Data for a whole QP sequence with constant matrices along with its optimal primal/dual solution vectors and the optimal objective function value is stored in plain ASCII files. For conveniently reading these files, three functions are provided (see <install-dir>/include/EXTRAS/OQPinterface.hpp for a detailed documentation):

- readOQPdimensions for reading the dimensions of the QP sequence,
- readOQPdata for reading data and solution information of the QP sequence,
- solveOQPbenchmark for solving a given benchmark QP sequence.

Moreover, the following function summarises the functionality of the three above-mentioned ones:

```
returnValue runOQPbenchmark( const char* path,
                             BooleanType isSparse,
                             BooleanType useHotstarts,
                             const Options& options,
                             int maxAllowedNWSR,
                             real_t& maxNWSR,
                             real_t& avgNWSR,
                             real_t& maxCPUtime,
                             real_t& avgCPUtime,
                             real_t& maxStationarity,
                             real_t& maxFeasibility,
                             real_t& maxComplementarity
                             );
```

It takes the path to the directory where the benchmark problem is stored as first argument. Second, the user can specify whether the QP matrices shall be converted to the sparse matrix format before solution and whether hotstarts shall be used when solving a QP sequence. Moreover, user-defined QP solver options and the maximum number of working set recalculations per QP are passed as input arguments. On output maxNWSR and avgNWSR contain the maximum and average number of working set recalculations, respectively, that have been actually performed, maxCPUtime and avgCPUtime contain the maximum and average CPU time, respectively, that have been required for solving each of the QPs. maxStationarity, maxFeasibility, maxComplementarity contain the maximum violations of the optimality conditions with respect to stationarity, feasibility and complementary of the obtained QP solutions, respectively.

For using this add-on you need to include the header file OQPinterface.hpp into your source code, a complete example can be found in the file <install-dir>/examples/example3.cpp. In order to run this example, you need to download example no. 01 from the Online QP Benchmark Collection website [1] first and extract its archive into the sub-folder <install-dir>/examples/chain80w/.

Chapter 6

Interfaces for Third-Party Software

If you want to use qpOASES via one of the following third-party interfaces, make sure that you have performed the installation steps 1 through 3 from Chapter 2. Afterwards, proceed with the installation of the desired interface as described in this chapter. Note that if you want use qpOASES solely from MATLAB, you can skip manual download and installation and obtain pre-compiled binaries by means of the `tbxManager` [10].

6.1 Interface for Matlab

Installation

It is possible to use qpOASES directly within the MATLAB environment. This is facilitated by compiling it into a so-called MEX function, which can be done as follows:

1. Start MATLAB and run `mex -setup` for choosing a C++ compiler (e.g. `gcc` or MICROSOFT® Visual Studio).

2. Execute the following commands:

   ```
   cd <install-dir>/interfaces/matlab
   make
   ```

   The latter command runs the MATLAB script `make.m` which does the compilation. Executables `qpOASES.<ext>` and `qpOASES_sequence.<ext>` should be created, where `<ext>` (e.g. `mexglx` or `mexw64`) depends on your operating system.

Remarks:

- The compilation was tested under both LINUX and WINDOWS using recent versions of MATLAB together with the gcc or the MICROSOFT® Visual Studio compiler. Still, modifications of the `make.m` script might be necessary depending on your operating system, your MATLAB version and your compiler.

- If compilation fails due to the fact that the `snprintf()` function is not supported, you might uncomment line 41 within `<install-dir>/include/qpOASES/Types.hpp` and try to compile again.

- The `make.m` script can be used in more advanced ways. Type `help make` for further information.

Interface for Solving a Single QP

After a successful installation, you can call qpOASES as conventional QP solver from the MATLAB environment (using a cold start every time):

```
[x,fval,exitflag,iter,lambda,auxOutput] = ...
                qpOASES( H,g,A,lb,ub,lbA,ubA{,options{,auxInput}} )
```

This command combines the creation of a QProblem object and a calls to the function init (see Chapter 3): the *input arguments*[1] specify the Hessian matrix, the gradient vector, the constraint matrix, the lower and upper bound vectors, the lower and upper constraints' vectors, respectively. Again, the Hessian has to be symmetric and positive semi-definite and may be left empty if you want to solve an LP (or a QP with identity Hessian matrix). All vectors must be stored as column vectors. It is possible to leave one or more of the input arguments lb, ub, lbA, ubA empty if your QP formulation does not comprise the corresponding limits. Optionally, structs containing options or auxiliary inputs can be passed.
Options can be generated using the qpOASES_options command. Called without arguments, it generates a struct containing all options as described in Section 5.2. For changing these values, two equivalent possibilities exist (see the MATLAB help for more details):

```
// change default values when creating options struct...
myOptions = qpOASES_options( 'printLevel',2, 'enableFlippingBounds',0 )

// or change them later
myOptions = qpOASES_options
myOptions.printLevel = 2
myOptions.enableFlippingBounds = 0
```

In addition to the options described in Section 5.2, the MATLAB options struct also contains the entries maxIter and maxCpuTime for specifying the maximum number of iterations and a CPU time limit, respectively. They correspond to nWSR and cputime in the C++ version. If maxIter is not set by the user, the default value $5 * (\mathtt{nV} + \mathtt{nC})$ is chosen. If maxCpuTime is not set, no additional CPU time limit is imposed.
An auxiliary input struct can be generated using the qpOASES_auxInput command. It allows the user to pass the following additional problem information:

- the Hessian type (e.g. for specifying a zero or identity Hessian matrix),
- an initial guess for the primal solution,
- an initial guess for the working sets of bounds and constraints (cf. Section 5.5),
- a pre-computed Cholesky factor of the Hessian matrix (see Section 5.7).

Please consult the MATLAB help for further details. If no initial guess is given, the usual homotopy starting at the origin is performed.

The *output arguments* contain the optimal primal solution vector, the optimal objective function value, a status flag, the number of iterations actually performed, and the optimal dual solution vector, respectively. The status flag can take one of the following values:

[1] matrices can be passed either in dense or sparse matrix format

- 0: QP was solved,
- 1: QP could not be solved within the given number of iterations,
- -1: QP could not be solved due to an internal error,
- -2: QP is infeasible and thus could not be solved,
- -3: QP is unbounded and thus could not be solved.

Moreover, the last output argument is a struct containing auxiliary outputs such as the (internally measured) *CPU time* or the internal *working set* at the solution. The working set is a subset of the active set corresponding to bound/constraint row vectors forming a linear independent set. The first nV elements correspond to the bounds, the last nC elements to the constraints. The working set is encoded as follows:

- 1: bound/constraint at its upper bound,
- 0: bound/constraint not at any bound,
- -1: bound/constraint at its lower bound.

If you do not need all output information, you can leave all but the first one away, e.g.

```
[x,fval] = qpOASES( H,g,A,lb,ub,[],ubA )
```

Remark: The function qpOASES also allows you to solve an *a priori* known sequence of QPs with fixed matrices: you just have to pass a whole sequence of input vectors. Each vector must be stored column-wise in a matrix, i.e. the ith QP is given by the ith columns of the QP "vectors" g, lb, ub, lbA, ubA, and all these five matrices must have the same number of columns. As both the Hessian and the constraint matrix remain constant, they are passed as in the case of a single QP. If a whole sequence of QPs is to be solved, also the outputs are given column-wise, i.e. x is a matrix with optimal primal solution vectors stored column-wise inside, fval is a row vector, and so on.

The interface allows you to directly use the QProblemB class for simply bounded QPs (cf. Section 4.3) by simply leaving the arguments A, lbA, ubA away:

```
[x,fval,exitflag,iter,lambda,auxOutput] =
                qpOASES( H,g,lb,ub{,options{,auxInput}} )
```

Again, a default value for the number of iterations is used (here $5 * \mathrm{nV}$) if maxIter is not specified within options. Also here you can leave lb or ub empty if they do not appear within your QP formulation.

Interface for Solving a QP Sequence

As the online active set strategy is intended to solve a whole sequence of parametrised QPs, there exist a special MATLAB function for "hot-starting" each QP from the solution of the previous one:

```
[QP,x,fval,...] = qpOASES_sequence( 'i',H,g,A,lb,ub,lbA,ubA{,options{,auxInput}} )
[x,fval,...]    = qpOASES_sequence( 'h',QP,g,lb,ub,lbA,ubA{,options} )
                  qpOASES_sequence( 'c',QP )
```

As in the C++ implementation (cf. Chapter 3), the first QP of the sequence is solved together with the initialisation all internal data structures. For this purpose, the function qpOASES_sequence (called with first input argument `'i'`) takes all QP data as well as, optionally, structs containing options or auxiliary inputs (see description above). Besides the usual output information, it provides one important additional output argument: the initialisation call returns as first output argument a handle to the QP object, which must be passed when hot-starting from this initial QP solution. Similar to the single QP case, you may leave all output arguments away except for the handle and the optimal primal solution vector.
Afterwards, each subsequent QP can be solved by performing a hot-start using the function qpOASES_sequence, again; this time called with first input argument `'h'` and by passing a handle to an QP object as second input argument. It takes the QP vectors of the new QP as well as, optionally, a set of options as further input arguments, and provides the usual output information.
Having solved the last QP of the sequence, you are encouraged to free the internal memory by calling `qpOASES_sequence( 'c',QP )`.

For solving QPs of special types as described in Chapter 4, special variants of the above function are provided: First, you can call

```
[QP,x,fval,...] = qpOASES_sequence( 'i',H,g,lb,ub{,options{,auxInput}} )
[x,fval,...]    = qpOASES_sequence( 'h',QP,g,lb,ub{,options} )
                  qpOASES_sequence( 'c',QP )
```

for solving simply bounded QPs (input arguments corresponding to constraints are simply left away). The internal memory is freed by calling `qpOASES_sequence( 'c',QP )`.
Second, call

```
[QP,x,fval,...] = qpOASES_sequence( 'i',H,g,A,lb,ub,lbA,ubA{,options{,auxInput}} )
[x,fval,...]    = qpOASES_sequence( 'm',QP,H,g,A,lb,ub,lbA,ubA{,options} )
                  qpOASES_sequence( 'c',QP )
```

for solving QPs with varying matrices, where the function qpOASES_sequence also takes the new matrices of the next QP of the sequence when called with first argument `'m'`.

Finally, the MATLAB interfaces offers to evaluate the local feedback law defined by the current active set. Call

```
[QP,x,fval,...] = qpOASES_sequence( 'i',H,g,A,lb,ub,lbA,ubA{,options{,auxInput}} )
[x,fval,...]    = qpOASES_sequence( 'e',QP,g,lb,ub,lbA,ubA{,options} )
                  qpOASES_sequence( 'c',QP )
```

for solving the equality constrained QP problem with constraints determined by the current active set. All bounds and inequality constraints which were not active in the previous solution might be violated. This command does not alter the internal state of qpOASES. Instead of calling this command multiple times, it is possible to supply several columns simultaneously in `g`, `lb`, `ub`, `lbA`, and `ubA`.

Examples

The files `example1.mat`, `example1a.mat` and `example1b.mat` contain, respectively, very basic examples for solving a sequence comprising two QPs with fixed matrices, varying matrices, and with simple bounds only. For solving the first one do the following:

1. Start MATLAB and execute the following commands:

   ```
   cd <install-dir>/interfaces/matlab
   load example1.mat
   ```

2. Solve the first QP by typing

   ```
   options = qpOASES_options( 'maxIter',10 );
   [QP,x,fval,exitflag,iter] =
          qpOASES_sequence( 'i',H,g,A,lb,ub,lbA,ubA,options )
   ```

3. Solve the second QP by typing

   ```
   [x,fval,exitflag,iter] =
          qpOASES_sequence( 'h',QP,g_new,lb_new,ub_new,lbA_new,ubA_new,options )
   ```

4. Free the internal memory by calling

   ```
   qpOASES_sequence( 'c',QP )
   ```

6.2 Interface for Simulink

Installation

You can use qpOASES directly within the SIMULINK environment, too. This requires to compile it into a so-called S function, which can be done as follows:

1. Start MATLAB and run `mex -setup` for choosing a C++ compiler (e.g. `gcc` or MICROSOFT® Visual Studio).

2. Execute the following commands:

   ```
   cd <install-dir>/interfaces/simulink
   make
   ```

   The latter command runs the MATLAB script `make.m` which does the compilation. Three executables called `qpOASES_QProblemB.<ext>`, `qpOASES_QProblem.<ext>` and `qpOASES_SQProblem.<ext>` should be created, where `<ext>` (e.g. `mexglx` or `mexw64`) depends on your operating system.

Remarks:

- The compilation was tested under both LINUX and WINDOWS using recent versions of MATLAB together with the gcc or the MICROSOFT® Visual Studio compiler. Still, modifications of the `make.m` script might be necessary depending on your operating system, your MATLAB version and your compiler.
- If compilation fails due to the fact that the `snprintf()` function is not supported, you might uncomment line 41 within `<install-dir>/include/qpOASES/Types.hpp` and try to compile again.
- The `make.m` script can be used in more advanced ways. Type `help make` for further information.

Interface

There exist three different S function interfaces corresponding to the three different types of QP sequences to be solved (see also Chapter 4):

1. `qpOASES_QProblemB.<ext>` for solving simply bounded QPs,
2. `qpOASES_QProblem.<ext>` for solving QPs with fixed matrices,
3. `qpOASES_SQProblem.<ext>` for solving QPs with varying matrices.

For each of these interfaces a simple example is provided within the folder `<install-dir>/interfaces/simulink`. We only give details for the one for QPs with fixed matrices, as the other ones work analogously.

In order to run the example, start MATLAB and execute the corresponding script file as follows:

```
cd <install-dir>/interfaces/simulink
load_example_QProblem
```

The sample QP data is loaded into the workspace and the file `qpOASES_QProblem.mdl` (depicted in Figure 6.1) is opened.

The `qpOASES` S function has *seven inputs*:

- the two (fixed) QP matrices `H` and `A` that are passed as S function parameters;
- the five QP vectors `g`, `lb`, `ub`, `lbA`, `ubA`, which are input signals and can be updated at each sampling instant.

The dimensions of the inputs are detected automatically, but they have to be consistent (e.g. the dimension of `H` needs to be the squared size of `g`). If the Hessian matrix `H` is the zero or identity matrix, the corresponding S function parameter can be left empty if the corresponding Hessian type is speficied in the S function code.

Moreover, you have to *define four additional values* in lines 57–60 of the file `qpOASES_QProblem.cpp` *before compilation* of the S function:

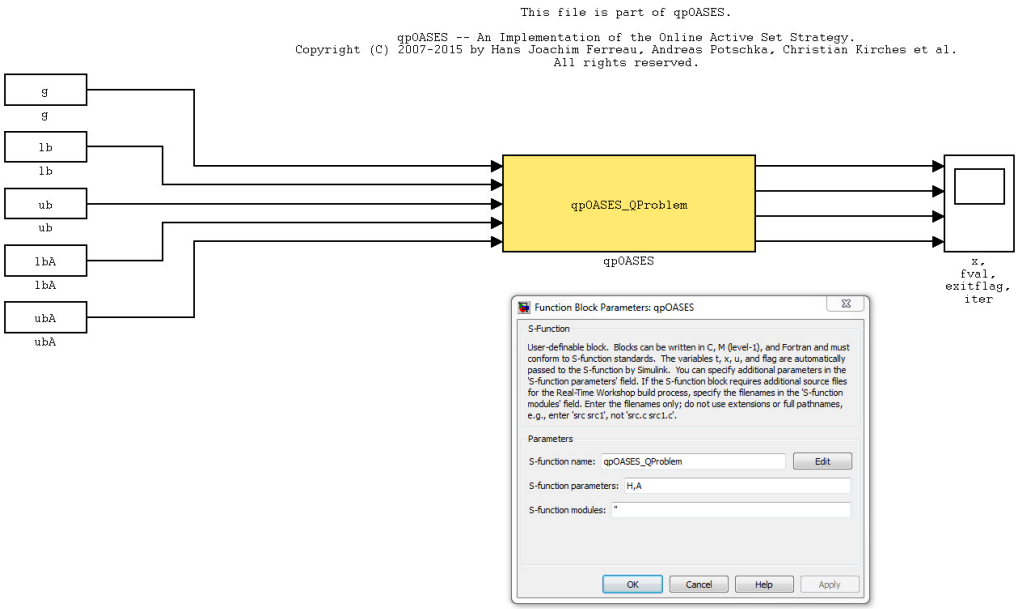


Figure 6.1: qpOASES working as SIMULINK S function.

- `#define SAMPLINGTIME <value>`: the sample time of the SIMULINK block;
- `#define NCONTROLINPUTS <value>`: the number of control inputs of your system (the leading `NCONTROLINPUTS` components of the optimal primal solution vector are returned as optimal output by the S function);
- `#define MAXITER <value>`: the maximum number of iterations to be performed per QP solution;
- `#define HESSIANTYPE <value>`: allows to pass additional information on the Hessian matrix when calling qpOASES. Choose `HST_ZERO` if you pass a zero matrix (i.e. if you intend to solve an LP) and `HST_IDENTITY` if the Hessian matrix is the identity matrix. Otherwise, the default choice `HST_UNKNOWN` will just work fine as qpOASES can also determine the correct type internally at the expense of a tiny runtime overhead.

For running the example you can use the specified default values; but do not forget to adjust them to the requirements of your own problem.

At each sampling instant the qpOASES S function provides the following *four outputs*[2]:

- `x`: the leading `NCONTROLINPUTS` components of optimal primal solution vector;
- `fval`: the optimal objective function value;
- `exitflag`: a status flag which can take one of the following values
  - 0: QP was solved,

[2]Note that the order of outputs has been slightly changed since release 3.0 in order to match the order of the MATLAB interface.

  - 1: QP could not be solved within the given number of working set recalculations,
  - -1: QP could not be solved due to an internal error,
  - -2: QP is infeasible and thus could not be solved,
  - -3: QP is unbounded and thus could not be solved;

- `iter`: the number of iterations actually performed.

An Example

Having executed the script `load_example_QProblem` as described above, you can simply start the SIMULINK simulation given by the file `example_QProblem.mdl`. The simulation runs for 0.5 s with a sample time of 0.1 s. At the first two sampling instants the QPs as specified in the file `example1.mat` of the MATLAB interface are solved; at the remaining sampling instants the last QP is solved repeatedly (requiring zero iterations as the hotstart feature of the online active set strategy is used).

6.3 Interface for Octave

qpOASES also provides an interface to OCTAVE that can be installed and used in the same way as the MATLAB interface. Thus, we refer to Section 6.1 for a description.

6.4 Interface for scilab

Note that the SCILAB interface features the main functionality of qpOASES, but is currently more limited and not fully compatible with the MATLAB interface!

Installation

For using qpOASES within SCILAB, you have to perform the following steps:

1. Compile the SCILAB interface by executing the following commands:

   ```
   cd <install-dir>/interfaces/scilab
   make
   ```

2. Start SCILAB and link the interface to the SCILAB environment:

   ```
   exec qpOASESinterface.sce;
   ```

Interface for Solving a Single QP

If you simply want to use qpOASES as conventional QP solver (using a cold start every time), you can call it as follows:

```
[x,fval,exitflag,iter,lambda] = qpOASES( H,g,A,lb,ub,lbA,ubA,nWSR )
```

The *input arguments* specify the Hessian matrix, the gradient vector, the constraint matrix, the lower and upper bound vectors, the lower and upper constraints' vectors, and the maximum number of working set recalculations, respectively. As usual, the Hessian must be symmetric and positive semi-definite and all vectors must be stored as column vectors. The *output arguments* contain the optimal primal solution vector, the optimal objective function value, a status flag, the number of iterations actually performed, and the optimal dual solution vector, respectively. The status flag can take one of the following values:

- 0: QP was solved,
- 1: QP could not be solved within the given number of working set recalculations,
- −1: QP could not be solved due to an internal error,
- −2: QP is infeasible and thus could not be solved,
- −3: QP is unbounded and thus could not be solved.

If you do not need all output information, you can leave all but the first one away.

Remark: A special variant for simply bounded QPs is not yet interfaced.

Interface for Solving a QP Sequence

As the online active set strategy is intended to solve a whole sequence of parametrised QPs, there exist special routines for doing so:

```
[x,fval,exitflag,iter,lambda] = qpOASES_init( H,g,A,lb,ub,lbA,ubA,nWSR )
[x,fval,exitflag,iter,lambda] = qpOASES_hotstart( g,lb,ub,lbA,ubA,nWSR )
                                qpOASES_cleanup
```

As in the C++ implementation (cf. Chapter 3), the first QP of the sequence is solved together with the initialisation all internal data structures. For this purpose, the function `qpOASES_init` takes all QP data and the maximum number of working set re-calculations for solving the initial QP as input arguments, and provides the usual output information (see above).
Afterwards, each subsequent QP is can be solved by performing a so-called "hot start" using the function `qpOASES_hotstart`. It takes the QP vectors of the new QP as well as the maximum number of working set re-calculations as input arguments, and provides the usual output information, again.
Having solved the last QP of the sequence, you are encouraged to free the internal memory by calling `qpOASES_cleanup`.

For solving QPs of special types as described in Chapter 4, special variants of the above functions are provided. First, the functions

```
[x,fval,exitflag,iter,lambda] = qpOASES_initSB( H,g,lb,ub,nWSR )
[x,fval,exitflag,iter,lambda] = qpOASES_hotstartSB( g,lb,ub,nWSR )
                                qpOASES_cleanupSB
```

for *s*imply *b*ounded QPs (input arguments corresponding to constraints are simply left away).

Second, the functions

```
[x,fval,exitflag,iter,lambda] = qpOASES_initVM( H,g,A,lb,ub,lbA,ubA,nWSR )
[x,fval,exitflag,iter,lambda] = qpOASES_hotstartVM( H,g,A,lb,ub,lbA,ubA,nWSR )
                                qpOASES_cleanupVM
```

for QPs with *v*arying *m*atrices, where qpOASES_hotstartVM also takes the new matrices of the next QP of the sequence.

Again, the internal memory is freed by calling qpOASES_cleanupSB and qpOASES_cleanupVM, respectively. This memory is kept independently for all three QP types.

Examples

The files example1.dat, example1a.dat and example1b.dat contain, respectively, very basic examples for solving a sequence comprising two QPs with fixed matrices, varying matrices, and with simple bounds only. For solving the first one do the following:

1. Start SCILAB and execute the following commands:

   ```
   cd <install-dir>/interfaces/scilab
   load('example1.dat')
   ```

2. Solve the first QP by typing

   ```
   [x,fval,exitflag,iter,lambda] =
             qpOASES_init( H,g,A,lb,ub,lbA,ubA,10 )
   ```

3. Solve the second QP by typing

   ```
   [x,fval,exitflag,iter,lambda] =
             qpOASES_hotstart( g_new,lb_new,ub_new,lbA_new,ubA_new,10 )
   ```

4. Free the internal memory by calling qpOASES_cleanup.

6.5 Interface for Python

Installation

Consult the file <install-dir>/interfaces/python/README.rst for a detailed description of the dependencies and installation options.

Using qpOASES from Python

The Python interface is a thin Cython wrapper of the C++ interface as defined in the header files <install-dir>/include/qpOASES. It strives to expose the contained functions and classes in an 1-to-1 fashion. The Python code resembles the C++ code and thus one can consult the C++ documentation for details.
In the the directory <install-dir>/interfaces/python/examples there are several basic examples, for instance <install-dir>/interfaces/python/examples/example1b.py

```
import numpy as np
from qpoases import PyQProblemB as QProblemB
from qpoases import PyBooleanType as BooleanType
from qpoases import PySubjectToStatus as SubjectToStatus
from qpoases import PyOptions as Options

# Setup data of first QP.

H   = np.array([1.0, 0.0, 0.0, 0.5 ]).reshape((2,2))
g   = np.array([1.5, 1.0 ])
lb  = np.array([0.5, -2.0])
ub  = np.array([5.0, 2.0 ])

# Setup data of second QP.

g_new   = np.array([1.0, 1.5])
lb_new  = np.array([0.0, -1.0])
ub_new  = np.array([5.0, -0.5])

# Setting up QProblemB object.

example = QProblemB(2)
options = Options()
options.enableFlippingBounds = BooleanType.FALSE
options.initialStatusBounds  = SubjectToStatus.INACTIVE
options.numRefinementSteps   = 1
example.setOptions(options)

# Solve first QP.

nWSR = 10
example.init(H, g, lb, ub, nWSR);
print("\nnWSR = %d\n\n"%nWSR)

# Solve second QP.

nWSR = 10;
example.hotstart(g_new, lb_new, ub_new, nWSR)
print("\nnWSR = %d\n\n"% nWSR)

# Get and print solution of second QP.

xOpt = np.zeros(2)
example.getPrimalSolution(xOpt)
print("\nxOpt = [ %e, %e ];  objVal = %e\n\n" %(xOpt[0], xOpt[1],
                                               example.getObjVal()))
```

One can run this script using

```
cd ./interfaces/python
python example1b.py
```

Using qpOASES in combination with Cython

Cython is a hybrid language between Python and C. It allows the user to

- call and interface to C/C++ code
- add static typing to Python code

The user writes `*.pyx` files instead of `*.py`, which are subsequently transformed to C code. This makes it possible to achieve the same performance as native C code.
In example `<install-dir>/interfaces/python/examples/cython/example1.pyx` we statically type the numpy arrays and access them in a nested loop. This contrived example would be relatively slow in pure Python, but not when compiled to native C code.

```
import numpy as np
from qpoases import PyQProblem as QProblem
from qpoases import PyPrintLevel as PrintLevel
from qpoases import PyOptions as Options
cimport numpy as np

def run():

    #Setup data of QP.

    cdef np.ndarray[np.double_t, ndim=2] H
    cdef np.ndarray[np.double_t, ndim=2] A
    cdef np.ndarray[np.double_t, ndim=1] g
    cdef np.ndarray[np.double_t, ndim=1] lb
    cdef np.ndarray[np.double_t, ndim=1] ub
    cdef np.ndarray[np.double_t, ndim=1] lbA
    cdef np.ndarray[np.double_t, ndim=1] ubA

    H   = np.array([1.0, 0.0, 0.0, 0.5 ]).reshape((2,2))
    A   = np.array([1.0, 1.0 ]).reshape((2,1))
    g   = np.array([1.5, 1.0 ])
    lb  = np.array([0.5, -2.0])
    ub  = np.array([5.0, 2.0 ])
    lbA = np.array([-1.0 ])
    ubA = np.array([2.0])

    # Setting up QProblem object.

    cdef example = QProblem(2, 1)
    cdef options = Options()
    options.printLevel = PrintLevel.NONE
    example.setOptions(options)

    # Solve first QP.

    cdef int nWSR = 10
    example.init(H, g, A, lb, ub, lbA, ubA, nWSR)
```

```
    # Solve subsequent QPs

    cdef int i,j
    for i in range(100000):
        for j in range(1, 100):
            g[0] = i%j
        example.hotstart(g, lb, ub, lbA, ubA, nWSR)

run()
```

Potential pitfalls and differences to the C++ version

- The Python interface uses `numpy.ndarrays` instead of raw pointers.
- No memory copies are performed, so you have to make sure that the numpy arrays you pass to `qpoases` are contiguous, have the correct size and order (row-major).
- Python does not feature enums. Instead, we use Python classes. Example: The C++-enum `returnValue`

  ```
  enum returnValue{
  ...
  RET_DIV_BY_ZERO,
  ...
  }
  ```

  is accessed in Python as static attribute of the class PyReturnValue

  ```
  import qpoases
  print(qpoases.PyReturnValue.DIV_BY_ZERO) # prints 1
  ```

  The leading `RET_` is omitted in the Python interface, since it is redundant .

Running the qpOASES unit test

There is a unit test suite for the Python interface of qpOASES. The tests are located in `<install-dir>/interfaces/python/tests/`. One can run the tests from the command line using `nosetests interfaces/python/tests`. Note that you may have to install the package `python-nose`.

6.6 Calling qpOASES from Plain C

qpOASES provides a wrapper that hides all C++ code in a library that can be linked against plain C code. See the folder `<install-dir>/interfaces/c` to find the code, a makefile as well as a couple of simple illustrating examples.
The wrapper allows the user to access qpOASES's main functionality but does not support all advanced features of Chapter 5. In particular, it only allows to instantiate a single QP object and does not support sparse matrices.

6.7 Running qpOASES on dSPACE

qpOASES can be easily run on a dSPACE board via its SIMULINK interface, provided that a C++ compiler is available. This has been tested for dSPACE boards version 5.3 or higher together with the dSPACE C++ Integration Kit 1.0.2 or higher. The following additional notes hopefully facilitate the setup:

1. Setup your dSPACE system.
2. Install the dSPACE C++ Integration Kit.
3. Install qpOASES (its SIMULINK interface, to be more precisely).
4. Compile qpOASES with compiler flag __DSPACE__. This can be done, e.g., by uncommenting line 45 within <install-dir>/include/qpOASES/Types.hpp.
5. Setup your SIMULINK project.
6. Open MK(make) file of your project (eventually you have to compile it once before) and add the following lines at the head of this file:

   ```
   # enable c++ support
   USER_BUILD_CPP_APPL = ON
   ```

7. Also complete the following lines:

   ```
   USER_SRCS = qpOASES_SQProblem.cpp qpOASES_QProblem.cpp
   qpOASES_QProblemB.cpp SQProblem.cpp QProblem.cpp QProblemB.cpp
   Bounds.cpp Constraints.cpp SubjectTo.cpp Indexlist.cpp
   Flipper.cpp Utils.cpp Options.cpp Matrices.cpp
   BLASReplacement.cpp LAPACKReplacement.cpp MessageHandling.cpp
   ```
   (i.e. all source files of qpOASES and its SIMULINK interface)

   ```
   USER_SRCS_DIR = ./src
   ```
   (i.e. directory of qpOASES source files)

   ```
   USER_INCLUDES_PATH = ./include ./src
   ```
   (i.e. directories of qpOASES header and source files)

8. Compile your project.
9. Run the compiled project on your dSPACE system.

6.8 Running qpOASES on xPC Target

qpOASES has also been run on xPC Target hardware via its SIMULINK interface. The following additional notes hopefully facilitate the setup[3]:

[3] Many thanks to Gergely Takacs for testing qpOASES on xPC Target hardware and for providing most of these hints.

1. Setup your xPC Target system.
2. Install qpOASES (its SIMULINK interface, to be more precisely).
3. Compile qpOASES with compiler flag __XPCTARGET__. This can be done, e.g., by uncommenting line 48 within <install-dir>/include/qpOASES/Types.hpp.
4. Setup your SIMULINK project.
5. Set simulation type in the Simulink scheme to "External".
6. Set system target file to "xpctarget.tlc" (via the menu Scheme→Simulation→Configuration parameters→Real-Time Workshop→System target file→xpctarget.tlc).
7. Integrate the <install-dir>/include directory and all source files into the SIMULINK scheme definition (via the menu Scheme→Simulation→Configuration parameters→Real-Time Workshop→Source Files→"../../include/" and "../../src/*.cpp").
8. Compile your project. (If you encounter troubles when re-compiling your project, try to first deleting directories produced by the Real-Time Workshop at the previous compilation cycle, namely slprj and <projectName>_xpc_rtw.)
9. Run the compiled project on your xPC Target hardware.

6.9 Using qpOASES within the ACADO Toolkit

ACADO TOOLKIT is a software framework for automatic control and dynamic optimisation available at

http://www.acadotoolkit.org.

It is an open-source (LGPL) environment for setting up a great variety of dynamic optimization problems for use in control, in particular (nonlinear) model predictive control [8]. ACADO TOOLKIT uses qpOASES as default QP solver, for linear MPC as well as for the QP sequences resulting from SQP-type methods.
An embedded variant of qpOASES is also used within the real-time NMPC algorithms as auto-generated by the ACADO CODE GENERATION tool [9].

6.10 Using qpOASES within MUSCOD-II

MUSCOD-II is a proprietary software package for numerical solution of optimal control problems involving differential-algebraic equations [3]. It has been developed by the members of the "Simulation and Optimization Group" of the Interdisciplinary Center for Scientific Computing (IWR) at University of Heidelberg. The current version of MUSCOD-II also contains an interface for using qpOASES as underlying QP solver.

6.11 Using qpOASES within YALMIP

qpOASES has been interfaced to YALMIP [11], a modelling language for solving convex and nonconvex optimization problems.

Chapter 7

Developer Information and Compiling Options

This chapter provides a very brief introduction to the qpOASES software design. If you are interested in using qpOASES within your own software project or in developing extensions for it yourself, we recommend to consult its DOXYGEN documentation (cf. installation step six described in Chapter 2) for detailed information. Moreover, you are encouraged to pose questions or remarks to `support@qpOASES.org`.

7.1 Class Hierarchy

So far, we mainly mentioned four different classes: `QProblem`, `QProblemB` `SQProblem` and `Options`. These are the only classes which provide user interfaces for accessing qpOASES's functionality. However, they are not the only classes of the qpOASES software package but are embedded in a more complex hierarchy.

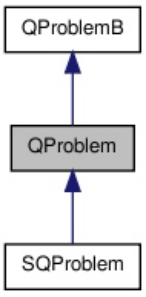


Figure 7.1: `QProblem` class hierarchy (illustrated with DOXYGEN [13]).

The class `QProblemB` is at the bottom of the hierarchy (see Figure 7.1) and provides all functionality necessary for solving a simply bounded quadratic program (cf. Section 4.3). The `QProblem` class is derived from it and implements all necessary additional functionality for solving a QPs comprising general constraints. The class `SQProblem`, in turn, inherits all features of the `QProblem` class and provides further functionality for handling QPs with varying matrices (cf. Section 4.2).

All the three classes QProblemB, QProblem and SQProblem make use of further auxiliary classes: First, they have a member of type Options to store user-defined QP solver options. Second, they hold members of type Bounds or Constraints (which are derived from a common type SubjectTo) in order to store bounds or constraints of a QP. Both the Bounds and the Constraints class manages lists (of type Indexlist) of free and fixed variables and active and inactive constraints, respectively. Third, they hold an instance of the Flipper class for storing a temporary copy of the matrix factorisations whenever necessary. Finally, they hold pointers to the matrices of the current QP and to a user-defined ConstraintProduct definition (see Section 5.4). QP matrices are stored within one of the classes depicted in Figure 7.2 depending on their structure.

All the above mentioned classes use a class called MessageHandling for providing errors messages, warnings or other information to the user and for handling return values of their member functions in a unified framework. This class makes use of the enumeration returnValue, which gathers all possible return values of all qpOASES functions. The current implementation uses a single *global* instance of the MessageHandling class; the global function

```
MessageHandling* getGlobalMessageHandler( );
```

returns a pointer to it.

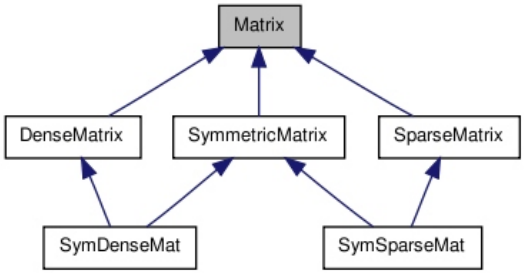


Figure 7.2: qpOASES matrix class hierarchy (illustrated with DOXYGEN [13]).

7.2 Global Constants

Some useful global constants are defined in file <install-dir>/include/Constants.hpp. Their default values seem to work reasonably, but you might change them if necessary:

- EPS: numerical value of machine precision,
- ZERO: numerical value of zero (for situations in which it would be unreasonable to compare with 0.0),
- INFTY: numerical value of infinity (e.g. for non-existing bounds),

7.3 Compiler Flags

When compiling qpOASES, you can define the following compiler flags:

- LINUX : activates LINUX-specific functionality (e.g. time measurement),
- WIN32 : activates WINDOWS-specific functionality (e.g. time measurement),
- __MATLAB__ : activates MATLAB-specific functionality (in particular, the use of mexPrintf instead of printf),
- __cplusplus : necessary for building C++ S functions for SIMULINK,
- __DSPACE__ : define this compiler flag in order to disable the qpOASES namespace (and switching off all text messages) for ensuring backward compatibility with dSPACE compilers,
- __XPCTARGET__ : define this compiler flag in order to disable all text messages for ensuring compatibility for xPC TARGET compilers,
- __NO_FMATH__ : define this compiler flag if no math library is available for your system in order to switch to custom implementations,
- __SINGLE_OBJECT__ : use this compiler flag to compile a qpOASES application into a single object file,
- __DEBUG__ : activates more detailed output messages during QP solution,
- __SUPPRESSANYOUTPUT__ : suppresses any console output during QP solution,
- __NO_COPYRIGHT__ : suppresses copyright notice at beginning of QP solution,
- __ALWAYS_INITIALISE_WITH_ALL_EQUALITIES__ : forces to always include all implicitly fixed bounds and all equality constraints into the initial working set when setting up an auxiliary QP,
- __USE_THREE_MULTS_GIVENS__ : switches to a different way of calculating Givens rotations that requires only three multiplications,
- __USE_SINGLE_PRECISION__ : switches to single precision arithmetic.

7.4 Unit Testing

qpOASES comes along with a basic unit testing for both the C++ version and the MATLAB interface. This section briefly describes how to run and extend them.

7.4.1 Testing the C++ Version

Unit tests for the C++ version can be found in the folder <install-dir>/testing/cpp and are supposed to be run under LINUX. Also tests for spotting potential memory leaks can be run if the VALGRIND tool has been installed on your system[1].

Running the Tests

In order to run the unit tests, perform the following steps:

1. *Compilation of all tests*:

   ```
   cd <install-dir>/testing
   make
   ```

2. *Run unit tests*:

   ```
   ./runUnitTests
   ```

   Each test reports whether it passed, failed or whether the corresponding test data has not been installed.

3. *Optional, check for potential memory leaks using* VALGRIND:

   ```
   ./checkForMemoryLeaks
   ```

   Each test reports whether memory leaks were found or not.

Adding New Tests

Use the following steps in order to add a new unit test:

1. *Setup a source file*:

   - Create a new source file with prefix `test_` within the folder <install-dir>/testing/cpp.
   - Include the header <install-dir>/include/qpOASES/UnitTesting.hpp and implement the main function in such a way that it returns `TEST_PASSED` if the test passed, `TEST_FAILED` if the test has failed, and `TEST_DATA_NOT_FOUND` if possibly required external data has not been found.
   - If required, store external test data within the folder <install-dir>/testing/cpp/data.

2. *Add test to Makefile and unit test shell script*:

   - Add the new source file to <install-dir>/testing/Makefile, i.e. add it to the list `QPOASES_TEST_EXES` and create a build target for its executable.
   - Call the new unit test within <install-dir>/testing/runUnitTests by adding the line

     ```
     runTest $counter ../bin/<yourTest>;
     ```

[1] See http://www.valgrind.org for further information.

7.4.2 Testing the Matlab Interface

Unit tests for the MATLAB interface can be found in the folder `<install-dir>/testing/matlab` and are supposed to be run under both LINUX and WINDOWS.

Running the Tests

In order to run the unit tests, perform the following steps:

1. *Start* MATLAB and change directory to

   `cd <install-dir>/testing/matlab`

2. *Run unit tests*:

   `runAllTests`

   which adds all sub-directories to the MATLAB path. Each test reports whether it passed, failed or whether the corresponding test data has not been installed.

Adding New Tests

Use the following steps in order to add a new unit test:

1. *Setup a* MATLAB *test script*:

   - Create a new test script within the folder `<install-dir>/testing/matlab/tests`.
   - Implement the script in such a way that it return a `successFlag` being `1` if the test passed, `0` if the test has failed, and `-1` if possibly required external data has not been found.
   - If required, store external test data within the folder `<install-dir>/testing/matlab/data`.

2. *Add test to main test script*: Call the new test script within

   `<install-dir>/testing/matlab/runAllTests.m`

   by adding the lines

   ```
   fprintf( 'Running <yourTest>...  ' )
   successFlag = updateSuccessFlag( successFlag, <yourTest>() );
   ```

Bibliography

[1] Online QP Benchmark Collection (backup), 2006-2012. http://www.qpOASES.org/onlineQP.

[2] M.J. Best. *Applied Mathematics and Parallel Computing*, chapter An Algorithm for the Solution of the Parametric Quadratic Programming Problem, pages 57–76. Physica-Verlag, Heidelberg, 1996.

[3] M. Diehl, D.B. Leineweber, and A.A.S. Schäfer. MUSCOD-II Users' Manual. IWR-Preprint 2001-25, Universität Heidelberg, 2001.

[4] H. J. Ferreau, H. G. Bock, and M. Diehl. An online active set strategy to overcome the limitations of explicit MPC. *International Journal of Robust and Nonlinear Control*, 18(8):816–830, 2008.

[5] H. J. Ferreau, P. Ortner, P. Langthaler, L. del Re, and M. Diehl. Predictive control of a real-world diesel engine using an extended online active set strategy. *Annual Reviews in Control*, 31(2):293–301, 2007.

[6] H.J. Ferreau. An Online Active Set Strategy for Fast Solution of Parametric Quadratic Programs with Applications to Predictive Engine Control. Master's thesis, University of Heidelberg, 2006.

[7] H.J. Ferreau, C. Kirches, A. Potschka, H.G. Bock, and M. Diehl. qpOASES: A parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation*, 6(4):327–363, 2014.

[8] B. Houska, H.J. Ferreau, and M. Diehl. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011.

[9] B. Houska, H.J. Ferreau, and M. Diehl. An Auto-Generated Real-Time Iteration Algorithm for Nonlinear MPC in the Microsecond Range. *Automatica*, 47(10):2279–2285, 2011.

[10] M. Kvasnica. tbxManager homepage. http://www.tbxmanager.com/, 2012–2014.

[11] J. Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.

[12] A. Potschka, C. Kirches, H.G. Bock, and J.P. Schlöder. Reliable solution of convex quadratic programs with parametric active set methods. Technical report, Interdisciplinary Center for Scientific Computing, Heidelberg University, Im Neuenheimer Feld 368, 69120 Heidelberg, GERMANY, November 2010.

[13] D. van Heesch. Doxygen homepage. http://www.stack.nl/~dimitri/doxygen/, 1997–2014.

Appendix A

qpOASES Software Licence

qpOASES is distributed under the terms of the GNU *Lesser* General Public License (LGPL) as published by the Free Software Foundation:

```
                    GNU LESSER GENERAL PUBLIC LICENSE
                         Version 2.1, February 1999

 Copyright (C) 1991, 1999 Free Software Foundation, Inc.
 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL.  It also counts
 as the successor of the GNU Library Public License, version 2, hence
 the version number 2.1.]

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

  This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it.  You
can use it too, but we suggest you first think carefully about whether
this license or the ordinary General Public License is the better
strategy to use in any particular case, based on the explanations below.

  When we speak of free software, we are referring to freedom of use,
not price.  Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and charge
for this service if you wish); that you receive source code or can get
it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do
these things.

  To protect your rights, we need to make restrictions that forbid
distributors to deny you these rights or to ask you to surrender these
rights.  These restrictions translate to certain responsibilities for
you if you distribute copies of the library or if you modify it.

  For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you.  You must make sure that they, too, receive or can get the source
```

```
code.  If you link other code with the library, you must provide
complete object files to the recipients, so that they can relink them
with the library after making changes to the library and recompiling
it.  And you must show them these terms so they know their rights.

  We protect your rights with a two-step method: (1) we copyright the
library, and (2) we offer you this license, which gives you legal
permission to copy, distribute and/or modify the library.

  To protect each distributor, we want to make it very clear that
there is no warranty for the free library.  Also, if the library is
modified by someone else and passed on, the recipients should know
that what they have is not the original version, so that the original
author's reputation will not be affected by problems that might be
introduced by others.

  Finally, software patents pose a constant threat to the existence of
any free program.  We wish to make sure that a company cannot
effectively restrict the users of a free program by obtaining a
restrictive license from a patent holder.  Therefore, we insist that
any patent license obtained for a version of the library must be
consistent with the full freedom of use specified in this license.

  Most GNU software, including some libraries, is covered by the
ordinary GNU General Public License.  This license, the GNU Lesser
General Public License, applies to certain designated libraries, and
is quite different from the ordinary General Public License.  We use
this license for certain libraries in order to permit linking those
libraries into non-free programs.

  When a program is linked with a library, whether statically or using
a shared library, the combination of the two is legally speaking a
combined work, a derivative of the original library.  The ordinary
General Public License therefore permits such linking only if the
entire combination fits its criteria of freedom.  The Lesser General
Public License permits more lax criteria for linking other code with
the library.

  We call this license the "Lesser" General Public License because it
does Less to protect the user's freedom than the ordinary General
Public License.  It also provides other free software developers Less
of an advantage over competing non-free programs.  These disadvantages
are the reason we use the ordinary General Public License for many
libraries.  However, the Lesser license provides advantages in certain
special circumstances.

  For example, on rare occasions, there may be a special need to
encourage the widest possible use of a certain library, so that it becomes
a de-facto standard.  To achieve this, non-free programs must be
allowed to use the library.  A more frequent case is that a free
library does the same job as widely used non-free libraries.  In this
case, there is little to gain by limiting the free library to free
software only, so we use the Lesser General Public License.

  In other cases, permission to use a particular library in non-free
programs enables a greater number of people to use a large body of
free software.  For example, permission to use the GNU C Library in
non-free programs enables many more people to use the whole GNU
operating system, as well as its variant, the GNU/Linux operating
system.

  Although the Lesser General Public License is Less protective of the
users' freedom, it does ensure that the user of a program that is
linked with the Library has the freedom and the wherewithal to run
```

that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

```
    c) You must cause the whole of the work to be licensed at no
    charge to all third parties under the terms of this License.

    d) If a facility in the modified Library refers to a function or a
    table of data to be supplied by an application program that uses
    the facility, other than as an argument passed when the facility
    is invoked, then you must make a good faith effort to ensure that,
    in the event an application does not supply such function or
    table, the facility still operates, and performs whatever part of
    its purpose remains meaningful.

    (For example, a function in a library to compute square roots has
    a purpose that is entirely well-defined independent of the
    application.  Therefore, Subsection 2d requires that any
    application-supplied function or table used by this function must
    be optional: if the application does not supply it, the square
    root function must still compute square roots.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Library,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Library, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote
it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Library.

In addition, mere aggregation of another work not based on the Library
with the Library (or with a work based on the Library) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library.  To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2,
instead of to this License.  (If a newer version than version 2 of the
ordinary GNU General Public License has appeared, then you can specify
that version instead if you wish.)  Do not make any other change in
these notices.

  Once this change is made in a given copy, it is irreversible for
that copy, so the ordinary GNU General Public License applies to all
subsequent copies and derivative works made from that copy.

  This option is useful when you wish to copy part of the code of
the Library into a program that is not a library.

  4. You may copy and distribute the Library (or a portion or
derivative of it, under Section 2) in object code or executable form
under the terms of Sections 1 and 2 above provided that you accompany
it with the complete corresponding machine-readable source code, which
must be distributed under the terms of Sections 1 and 2 above on a
medium customarily used for software interchange.

  If distribution of object code is made by offering access to copy
```

from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the

Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and

all its terms and conditions for copying, distributing or modifying
the Library or works based on it.

  10. Each time you redistribute the Library (or any work based on the
Library), the recipient automatically receives a license from the
original licensor to copy, distribute, link with or modify the Library
subject to these terms and conditions. You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties with
this License.

  11. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License. If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Library at all. For example, if a patent
license would not permit royalty-free redistribution of the Library by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any
particular circumstance, the balance of the section is intended to apply,
and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system which is
implemented by public license practices. Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  12. If the distribution and/or use of the Library is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Library under this License may add
an explicit geographical distribution limitation excluding those countries,
so that distribution is permitted only in or among countries not thus
excluded. In such case, this License incorporates the limitation as if
written in the body of this License.

  13. The Free Software Foundation may publish revised and/or new
versions of the Lesser General Public License from time to time.
Such new versions will be similar in spirit to the present version,
but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library
specifies a version number of this License which applies to it and
"any later version", you have the option of following the terms and
conditions either of that version or of any later version published by
the Free Software Foundation. If the Library does not specify a
license version number, you may choose any version ever published by
the Free Software Foundation.

  14. If you wish to incorporate parts of the Library into other free

programs whose distribution conditions are incompatible with these,
write to the author to ask for permission. For software which is
copyrighted by the Free Software Foundation, write to the Free
Software Foundation; we sometimes make exceptions for this. Our
decision will be guided by the two goals of preserving the free status
of all derivatives of our free software and of promoting the sharing
and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO
WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW.
EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR
OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY
KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE
LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME
THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN
WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY
AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU
FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE
LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING
RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A
FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF
SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest
possible use to the public, we recommend making it free software that
everyone can redistribute and change. You can do so by permitting
redistribution under these terms (or, alternatively, under the terms of the
ordinary General Public License).

To apply these terms, attach the following notices to the library. It is
safest to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least the
"copyright" line and a pointer to where the full notice is found.

    <one line to give the library's name and a brief idea of what it does.>
    Copyright (C) <year> <name of author>

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

```
You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the library, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the
  library ‘Frob’ (a library for tweaking knobs) written by James Random Hacker.

  <signature of Ty Coon>, 1 April 1990
  Ty Coon, President of Vice

That’s all there is to it!
```