OSQP Documentation

Release 0.2.1

Bartolomeo Stellato, Goran Banjac

Jan 23, 2018

User Documentation

Join our forum **for any questions related to the solver!**

The OSQP (Operator Splitting Quadratic Program) solver is a numerical optimization package for solving convex quadratic programs in the form

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T Px + q^T x \\ \text{subject to} & l \leq Ax \leq u \end{array}$$

where x is the optimization variable and $P \in \mathbf{S}_+^n$ a positive semidefinite matrix.

Code available on GitHub.

Citing OSQP

If you are using OSQP for your work, we encourage you to

- *Cite the related papers*
- Put a star on GitHub

We are looking forward to hearing your success stories with OSQP! Please share them with us.

Features

Efficient It uses a custom ADMM-based first-order method requiring only a single matrix factorization in the setup phase. All the other operations are extremely cheap. It also implements custom sparse linear algebra routines exploiting structures in the problem data.

Robust The algorithm is absolutely division free after the setup and it requires no assumptions on problem data (the problem only needs to be convex). It just works!

Detects primal / dual infeasible problems When the problem is primal or dual infeasible, OSQP detects it. It is the first available QP solver based on first-order methods able to do so.

Embeddable It has an easy interface to generate customized embeddable C code with no memory manager required.

Library-free It requires no external library to run. Only the setup phase requires the AMD and SparseLDL from Timothy A. Davis that are already included in the sources.

Efficiently warm started It can be easily warm-started and the matrix factorization can be cached to solve parametrized problems extremely efficiently.

Interfaces It can be interfaced to C, C++, Python, Julia and Matlab.

License

OSQP is distributed under the Apache 2.0 License

Credits

The following people have been involved in the development of OSQP:

- Bartolomeo Stellato (University of Oxford): main development
- Goran Banjac (University of Oxford): main development
- Nicholas Moehle (Stanford University): methods, maths, and code generation

- Paul Goulart (University of Oxford): methods, maths, and Matlab interface
- Alberto Bemporad (IMT Lucca): methods and maths
- Stephen Boyd (Stanford University): methods and maths

Bug reports and support

Please report any issues via the Github issue tracker. All types of issues are welcome including bug reports, documentation typos, feature requests and so on.

Numerical benchmarks

Numerical benchmarks against other solvers are available here.

CHAPTER 1

The solver

1.1 Problem statement

OSQP solves convex quadratic programs (QPs) of the form

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T P x + q^T x \\ \text{subject to} & l \leq Ax \leq u \end{array}$$

where $x \in \mathbf{R}^n$ is the optimization variable. The objective function is defined by a positive semidefinite matrix $P \in \mathbf{S}_+^n$ and vector $q \in \mathbf{R}^n$. The linear constraints are defined by matrix $A \in \mathbf{R}^{m \times n}$ and vectors $l \in \mathbf{R}^m \cup \{-\infty\}^m$, $u \in \mathbf{R}^m \cup \{+\infty\}^m$.

1.2 Algorithm

The solver runs the following ADMM algorithm (for more details see the related papers at the *Citing OSQP* section):

$$\begin{aligned} (x^{k+1}, \nu^{k+1}) &\leftarrow \text{solve linear system} \\ \tilde{z}^{k+1} &\leftarrow z^k + \rho^{-1}(\nu^{k+1} - y^k) \\ z^{k+1} &\leftarrow \Pi(\tilde{z}^k + \rho^{-1} y^k) \\ y^{k+1} &\leftarrow y^k + \rho(\tilde{z}^{k+1} - z^{k+1}) \end{aligned}$$

where Π is the projection onto the hyperbox $[l, u]$. ρ is the ADMM step-size.

1.2.1 Linear system solution

The linear system solution is the core part of the algorithm. It can be done using a **direct** or **indirect** method.

With a **direct linear system solver** we solve the following linear system with a quasi-definite matrix

$$\begin{bmatrix} P + \sigma I & A^T \\ A & -\rho^{-1} I \end{bmatrix} \begin{bmatrix} x^{k+1} \\ \nu^{k+1} \end{bmatrix} = \begin{bmatrix} \sigma x^k - q \\ z^k - \rho^{-1} y^k \end{bmatrix}.$$

With an **indirect linear system solver** we solve the following linear system with a positive definite matrix

$$\left(P + \sigma I + \rho A^T A\right) x^{k+1} = \sigma x^k - q + A^T(\rho z^k - y^k).$$

OSQP core is designed to support different linear system solvers. For their installation see *this section*. To specify your favorite linear system solver see *this section*.

1.2.2 Convergence

At each iteration k OSQP produces a tuple (x^k, z^k, y^k) with $x^k \in \mathbf{R}^n$ and $z^k, y^k \in \mathbf{R}^m$.

The primal and and dual residuals associated to (x^k, z^k, y^k) are

$$r^k_{\text{prim}} = Ax^k - z^k$$
$$r^k_{\text{dual}} = Px^k + q + A^T y^k.$$

Complementary slackness is satisfied by construction at machine precision. If the problem is feasible, the residuals converge to zero as $k \to \infty$. The algorithm stops when the norms of r^k_{prim} and r^k_{dual} are within the specified tolerance levels $\epsilon_{\text{prim}} > 0$ and $\epsilon_{\text{dual}} > 0$

$$\|r^k_{\text{prim}}\|_\infty \leq \epsilon_{\text{prim}}, \quad \|r^k_{\text{dual}}\|_\infty \leq \epsilon_{\text{dual}}.$$

We set the tolerance levels as

$$\epsilon_{\text{prim}} = \epsilon_{\text{abs}} + \epsilon_{\text{rel}} \max\{\|Ax^k\|_\infty, \|z^k\|_\infty\}$$
$$\epsilon_{\text{dual}} = \epsilon_{\text{abs}} + \epsilon_{\text{rel}} \max\{\|Px^k\|_\infty, \|A^T y^k\|_\infty, \|q\|_\infty\}.$$

1.2.3 ρ step-size

To ensure quick convergence of the algorithm we adapt ρ by balancing the residuals. In default mode, the inteval (*i.e.*, number of iterations) at which we update rho is defined by a time measurement. When the iterations time becomes greater than a certain fraction of the setup time, *i.e.* `adaptive_rho_fraction`, we set the current number of iterations as the interval to update ρ. The update happens as follows

$$\rho^{k+1} \leftarrow \rho^k \sqrt{\frac{\|r_{\text{prim}}\|_\infty}{\|r_{\text{dual}}\|_\infty}}.$$

Note that ρ is updated only if it is sufficiently different than the current one. In particular if it is `adaptive_rho_tolerance` times larger or smaller than the current one.

1.3 Infeasible problems

OSQP is able to detect if the problem is primal or dual infeasible.

1.3.1 Primal infeasibility

When the problem is primal infeasible, the algorithm generates a certificate of infeasibility, *i.e.*, a vector $v \in \mathbf{R}^m$ such that

$$A^T v = 0, \quad u^T v_+ + l^T v_- < 0,$$

where $v_+ = \max(v, 0)$ and $v_- = \min(v, 0)$.

The primal infeasibility check is then

$$\left\|A^T v\right\|_\infty \leq \epsilon_{\text{prim\_inf}}, \quad u^T (v)_+ + l^T (v)_- \leq -\epsilon_{\text{prim\_inf}}.$$

1.3.2 Dual infeasibility

When the problem is dual infeasible, OSQP generates a vector $s \in \mathbf{R}^n$ being a certificate of dual infeasibility, *i.e.*,

$$Ps = 0, \quad q^T s < 0, \quad (As)_i = \begin{cases} 0 & l_i \in \mathbf{R}, u_i \in \mathbf{R} \\ \geq 0 & l_i \in \mathbf{R}, u_i = +\infty \\ \leq 0 & u_i \in \mathbf{R}, l_i = -\infty. \end{cases}$$

The dual infeasibility check is then

$$\|Ps\|_\infty \leq \epsilon_{\text{dual\_inf}}, \quad q^T s \leq -\epsilon_{\text{dual\_inf}},$$
$$(As)_i \begin{cases} \in [-\epsilon_{\text{dual\_inf}}, \epsilon_{\text{dual\_inf}}] & u_i, l_i \in \mathbf{R} \\ \geq \epsilon_{\text{dual\_inf}} & u_i = +\infty \\ \leq -\epsilon_{\text{dual\_inf}} & l_i = -\infty. \end{cases}$$

1.3.3 Polishing

Polishing is an additional algorithm step where OSQP tries to compute a high-accuracy solution. We can enable it by turning the setting `polish` to 1.

Polishing works by guessing the active constraints at the optimum and solving an additional linear system. If the guess is correct, OSQP returns a high accuracy solution. Otherwise OSQP returns the ADMM solution. The status of the polishing phase appears in the information `status_polish`.

Note that polishing requires the solution of an additional linear system and thereby, an additional factorization if the linear system solver is direct. However, the linear system is usually much smaller than the one solved during the ADMM iterations.

The chances to have a successful polishing increase if the tolerances `eps_abs` and `eps_rel` are small. However, low tolerances might require a very large number of iterations.

CHAPTER 2

Installation

2.1 Build from sources

2.1.1 Install GCC and CMake

The main compilation directives are specified using

- GCC compiler to build the binaries
- CMake to create the Makefiles

Linux

Both `gcc` and `cmake` commands are already installed by default.

Mac OS

Install Xcode and command line tools

1. Install the latest release of Xcode.
2. Install the command line tools by executing from the terminal

```
xcode-select --install
```

Install CMake via Homebrew

1. Install Homebrew and update its packages to the latest version.
2. Install cmake by executing

```
brew install cmake
```

Windows

1. Install TDM-GCC 32bit or 64bit depending on your platform.
2. Install the latest binaries of CMake.
3. Make sure you have the privileges to create symbolic links. See the git wiki for more details.

2.1.2 Build the binaries

Run the following commands from the terminal

1. Clone the repository, create `build` directory and change directory
   - On Linux and Mac OS run

     ```
     git clone https://github.com/oxfordcontrol/osqp
     cd osqp
     mkdir build
     cd build
     ```

   - On Windows run

     ```
     git clone -c core.symlinks=true https://github.com/oxfordcontrol/osqp
     cd osqp
     mkdir build
     cd build
     ```

2. Create Makefiles
   - In Linux and Mac OS run

     ```
     cmake -G "Unix Makefiles" ..
     ```

   - In Windows run

     ```
     cmake -G "MinGW Makefiles" ..
     ```

3. Compile OSQP

   ```
   cmake --build .
   ```

Thanks to CMake, it is possible to create projects for a wide variety of IDEs; see here for more details. For example, to create a project for Visual Studio 14 2015, it is just necessary to run

```
cmake -G "Visual Studio 14 2015" ..
```

The compilation will generate the demo `osqp_demo` and the unittests `osqp_tester` executables. In the case of `Unix` or `MinGW Makefiles` option they are located in the `build/out/` directory. Run them to check that the compilation was correct.

Once the sources are built, the generated static `build/out/libosqpstatic.a` and shared `build/out/libosqp.ext` libraries can be used to interface any C/C++ software to OSQP (see *CC++ interface* installation).

2.2 CC++ interface

2.2.1 Binaries

Precompiled platform-dependent shared and static libraries are available on Bintray. We here assume that the user uncompressed each archive to `OSQP_FOLDER`.

Each archive contains static `OSQP_FOLDER/lib/libosqpstatic.a` and shared `OSQP_FOLER/lib/libosqp.ext` libraries to be used to interface OSQP to any C/C++ software. The required include files can be found in `OSQP_FOLDER/include`.

Simply compile with the linker option with `-LOSQP_FOLDER/lib` and `-losqp` or `-losqpstatic`.

2.2.2 Sources

The OSQP libraries can also be compiled from sources. For more details see *Build from sources*.

2.3 Python

Python interface supports Python 2.7 and 3.5 or newer.

2.3.1 Anaconda

```
conda install -c oxfordcontrol osqp
```

2.3.2 Pip

```
pip install osqp
```

2.3.3 Sources

You need to install the following (see *Build from sources* for more details):

- GCC compiler
- CMake

Note: **Windows**: You need to install **also** the Visual Studio C++ compiler:

- Python 2: Visual C++ 9.0 for Python (VC 9.0)
- Python 3: Visual C++ 2015 Build Tools (VC 14.0)

Now you are ready to build OSQP python interface from sources. Run the following in your terminal

```
git clone https://github.com/oxfordcontrol/osqp
cd osqp/interfaces/python
python setup.py install
```

2.4 Julia

Julia interface can be directly installed from Julia package manager by running

```
Pkg.add("OSQP")
```

The interface code is available on GitHub.

2.5 Matlab

OSQP Matlab interface requires Matlab 2015b or newer.

2.5.1 Binaries

Precompiled platform-dependent Matlab binaries are available on Bintray.

To install the interface, just run the following commands:

```
websave('install_osqp.m', 'https://dl.bintray.com/bstellato/generic/OSQP/0.2.1/install_osqp.m');
install_osqp
```

2.5.2 Sources

You need to install the following (see *Build from sources* for more details):

- A supported 64bit C/C++ compiler
- CMake

After you install both, check that your compiler is selected by executing

```
mex -setup
```

Note: **Windows**: If Matlab does not find TDM-GCC, you need to set the environment variable `MW_MINGW64_LOC` as follows

```
setenv('MW_MINGW64_LOC', 'C:\TDM-GCC-64')
```

where `C:\TDM-GCC-64` is the installation folder for TDM-GCC.

You can now build the interface by running inside Matlab

```
!git clone https://github.com/oxfordcontrol/osqp
cd osqp/interfaces/matlab
make_osqp
```

Then you can add the interface to the search path by executing from the same directory

```
addpath(pwd)
savepath
```

2.6 Linear System Solvers

The linear system solver is a core part of the OSQP algorithm. Depending on the problem instance, different linear system solvers can greatly speedup or reduce the computation time of OSQP. To set the preferred linear system solver, see *Linear Systems Solvers*.

2.6.1 Dynamic shared library loading

OSQP dynamically loads the shared libraries related to the selected external solver. Thus, there is no need to link it at compile time. The only requirement is that the shared library related to the solver is in the library path of the operating system

Operating system	Path variable	Extension
Linux	`LD_LIBRARY_PATH`	`.so`
Mac	`DYLD_LIBRARY_PATH`	`.dylib`
Windows	`PATH`	`.dll`

2.6.2 SuiteSparse LDL

OSQP comes with SuiteSparse LDL internally installed. It does not require any external shared library. SuiteSparse LDL is a sparse direct solver that works well for most small to medium sized problems. However, its becomes not really efficient for large scale problems since it is not multi-threaded.

2.6.3 MKL Pardiso

MKL Pardiso is an efficient multi-threaded linear system solver that works well for large scale problems part of the Intel Math Kernel Library. Intel offers free lincenses for MKL for most non-commercial applications.

Install with MKL

We can install MKL Pardiso by using the standard MKL installer. The main library to be loaded is called `libmkl_rt`. To add it, together with its dependencies, to your path, just execute the automatic MKL script.

Operating system	Script
Linux	`source $MKLROOT/bin/mklvars.sh intel64`
Mac	`source $MKLROOT/bin/mklvars.sh intel64`
Windows	`%MKLROOT%\mklvars.bat intel64`

where `MKLROOT` is the MKL installation directory.

Install with Anaconda

Anaconda Python distribution comes with the intel MKL libraries preinstalled including MKL Pardiso. To use this version, the Anaconda libraries folders have to be added to the system path. In particular, given the Anaconda installation directory `ANACONDA_ROOT`, we need to add the following folders:

- `libmkl_rt`: This library is located in the folder `ANACONDA_ROOT/pkgs/mkl-VERSION/lib` where `VERSION` is the version of the `mkl` package.
- `libiomp`: This library is located in the folder `ANACONDA_ROOT/pkgs/intel-openmp-VERSION/lib` where `VERSION` is the version of the `intel-openmp` package.

CHAPTER 3

Interfaces

3.1 C/C++

3.1.1 Main solver API

The main C/C++ API is imported from the header `osqp.h` and provides the following functions

OSQPWorkspace ***osqp_setup**(**const** *OSQPData* *data, *OSQPSettings* *settings)
Initialize OSQP solver allocating memory.

All the inputs must be already allocated in memory before calling.

It performs:

- data and settings validation
- problem data scaling
- automatic parameters tuning (if enabled)
- setup linear system solver:
  - direct solver: KKT matrix factorization is performed here
  - indirect solver: KKT matrix preconditioning is performed here

NB: This is the only function that allocates dynamic memory and is not used during code generation

Return Solver environment

Parameters

- `data`: Problem data
- `settings`: Solver settings

c_int **osqp_solve**(*OSQPWorkspace* *work)
Solve quadratic program

The final solver information is stored in the *work->info* structure

The solution is stored in the *work->solution* structure

If the problem is primal infeasible, the certificate is stored in *work->delta_y*

If the problem is dual infeasible, the certificate is stored in *work->delta_x*

Return Exitflag for errors

Parameters

- `work`: Workspace allocated

c_int **osqp_cleanup**(*OSQPWorkspace* **work*)
Cleanup workspace by deallocating memory

This function is not used in code generation

Return Exitflag for errors

Parameters

- `work`: Workspace

3.1.2 Sublevel API

Sublevel C/C++ API is also imported from the header `osqp.h` and provides the following functions

Warm start

OSQP automatically warm starts primal and dual variables from the previous QP solution. If you would like to warm start their values manually, you can use

c_int **osqp_warm_start**(*OSQPWorkspace* **work*, c_float **x*, c_float **y*)
Warm start primal and dual variables

Return Exitflag

Parameters

- `work`: Workspace structure
- `x`: Primal variable
- `y`: Dual variable

c_int **osqp_warm_start_x**(*OSQPWorkspace* **work*, c_float **x*)
Warm start primal variable

Return Exitflag

Parameters

- `work`: Workspace structure
- `x`: Primal variable

c_int **osqp_warm_start_y**(*OSQPWorkspace* **work*, c_float **y*)
Warm start dual variable

Return Exitflag

Parameters

- `work`: Workspace structure
- `y`: Dual variable

Update problem data

Problem data can be updated without executing the setup again using the following functions.

c_int **`osqp_update_lin_cost`**(*OSQPWorkspace* *work, c_float *q_new)
Update linear cost in the problem

Return Exitflag for errors and warnings

Parameters

- `work`: Workspace
- `q_new`: New linear cost

c_int **`osqp_update_lower_bound`**(*OSQPWorkspace* *work, c_float *l_new)
Update lower bound in the problem constraints

Return Exitflag: 1 if new lower bound is not <= than upper bound

Parameters

- `work`: Workspace
- `l_new`: New lower bound

c_int **`osqp_update_upper_bound`**(*OSQPWorkspace* *work, c_float *u_new)
Update upper bound in the problem constraints

Return Exitflag: 1 if new upper bound is not >= than lower bound

Parameters

- `work`: Workspace
- `u_new`: New upper bound

c_int **`osqp_update_bounds`**(*OSQPWorkspace* *work, c_float *l_new, c_float *u_new)
Update lower and upper bounds in the problem constraints

Return Exitflag: 1 if new lower bound is not <= than new upper bound

Parameters

- `work`: Workspace
- `l_new`: New lower bound
- `u_new`: New upper bound

c_int **`osqp_update_P`**(*OSQPWorkspace* *work, c_float *Px_new, c_int *Px_new_idx, c_int P_new_n)
Update elements of matrix P (upper-diagonal) without changing sparsity structure.

If Px_new_idx is OSQP_NULL, Px_new is assumed to be as long as P->x and the whole P->x is replaced.

Update elements of matrix P (upper-diagonal) without changing sparsity structure.

Return output flag: 0: OK 1: P_new_n > nnzP <0: error in the update

Parameters

- `work`: Workspace structure
- `Px_new`: Vector of new elements in P->x (upper triangular)

- `Px_new_idx`: Index mapping new elements to positions in P->x
- `P_new_n`: Number of new elements to be changed

If Px_new_idx is OSQP_NULL, Px_new is assumed to be as long as P->x and the whole P->x is replaced.

Return output flag: 0: OK 1: P_new_n > nnzP <0: error in update_matrices()

Parameters

- `work`: Workspace structure
- `Px_new`: Vector of new elements in P->x (upper triangular)
- `Px_new_idx`: Index mapping new elements to positions in P->x
- `P_new_n`: Number of new elements to be changed

c_int **osqp_update_A**(*OSQPWorkspace* *work, c_float *Ax_new, c_int *Ax_new_idx, c_int A_new_n)

Update elements of matrix A without changing sparsity structure.

If Ax_new_idx is OSQP_NULL, Ax_new is assumed to be as long as A->x and the whole P->x is replaced.

Update elements of matrix A without changing sparsity structure.

Return output flag: 0: OK 1: A_new_n > nnzA <0: error in the update

Parameters

- `work`: Workspace structure
- `Ax_new`: Vector of new elements in A->x
- `Ax_new_idx`: Index mapping new elements to positions in A->x
- `A_new_n`: Number of new elements to be changed

If Ax_new_idx is OSQP_NULL, Ax_new is assumed to be as long as A->x and the whole P->x is replaced.

Return output flag: 0: OK 1: A_new_n > nnzA <0: error in update_matrices()

Parameters

- `work`: Workspace structure
- `Ax_new`: Vector of new elements in A->x
- `Ax_new_idx`: Index mapping new elements to positions in A->x
- `A_new_n`: Number of new elements to be changed

c_int **osqp_update_P_A**(*OSQPWorkspace* *work, c_float *Px_new, c_int *Px_new_idx, c_int P_new_n, c_float *Ax_new, c_int *Ax_new_idx, c_int A_new_n)

Update elements of matrix P (upper-diagonal) and elements of matrix A without changing sparsity structure.

If Px_new_idx is OSQP_NULL, Px_new is assumed to be as long as P->x and the whole P->x is replaced.

If Ax_new_idx is OSQP_NULL, Ax_new is assumed to be as long as A->x and the whole P->x is replaced.

Update elements of matrix P (upper-diagonal) and elements of matrix A without changing sparsity structure.

Return output flag: 0: OK 1: P_new_n > nnzP 2: A_new_n > nnzA <0: error in the update

Parameters

- `work`: Workspace structure
- `Px_new`: Vector of new elements in P->x (upper triangular)

- `Px_new_idx`: Index mapping new elements to positions in P->x
- `P_new_n`: Number of new elements to be changed
- `Ax_new`: Vector of new elements in A->x
- `Ax_new_idx`: Index mapping new elements to positions in A->x
- `A_new_n`: Number of new elements to be changed

If Px_new_idx is OSQP_NULL, Px_new is assumed to be as long as P->x and the whole P->x is replaced.

If Ax_new_idx is OSQP_NULL, Ax_new is assumed to be as long as A->x and the whole P->x is replaced.

Return output flag: 0: OK 1: P_new_n > nnzP 2: A_new_n > nnzA <0: error in update_matrices()

Parameters

- `work`: Workspace structure
- `Px_new`: Vector of new elements in P->x (upper triangular)
- `Px_new_idx`: Index mapping new elements to positions in P->x
- `P_new_n`: Number of new elements to be changed
- `Ax_new`: Vector of new elements in A->x
- `Ax_new_idx`: Index mapping new elements to positions in A->x
- `A_new_n`: Number of new elements to be changed

3.1.3 Data types

The most basic used datatypes are

- `c_int`: can be `long` or `int` if the compiler flag `DLONG` is set or not
- `c_float`: can be a `float` or a `double` if the compiler flag `DFLOAT` is set or not.

The relevant structures used in the API are

Data

struct OSQPData
Data structure

Public Members

c_int **n**
number of variables n,

c_int **m**
number of constraints m

csc ***P**
P: in csc format (size n x n). The workspace version stores only the upper triangular part. P->nzmax is the number of nonzero elements of the full P.

csc ***A**
A: in csc format (size m x n)

c_float *q
dense array for linear part of cost function (size n)

c_float *l
dense array for lower bound (size m)

c_float *u
dense array for upper bound (size m)

The matrices are defined in Compressed Sparse Column (CSC) format.

struct csc
Matrix in compressed-column or triplet form

Public Members

c_int nzmax
maximum number of entries.

c_int m
number of rows

c_int n
number of columns

c_int *p
column pointers (size n+1) (col indices (size nzmax) start from 0 when using triplet format (direct KKT matrix formation))

c_int *i
row indices, size nzmax starting from 0

c_float *x
numerical values, size nzmax

c_int nz
of entries in triplet matrix, -1 for csc

Settings

struct OSQPSettings
Settings struct

Public Members

c_float rho
ADMM step rho.

c_float sigma
ADMM step sigma.

c_int scaling
heuristic data scaling iterations. If 0, scaling disabled

c_int adaptive_rho
boolean, is rho step size adaptive?

c_int adaptive_rho_interval
Number of iterations between rho adaptations rho. If 0, it is automatic.

c_float **adaptive_rho_tolerance**
: Tolerance X for adapting rho. The new rho has to be X times larger or 1/X times smaller than the current one to trigger a new factorization.

c_float **adaptive_rho_fraction**
: Interval for adapting rho (fraction of the setup time)

c_int **max_iter**
: maximum iterations

c_float **eps_abs**
: absolute convergence tolerance

c_float **eps_rel**
: relative convergence tolerance

c_float **eps_prim_inf**
: primal infeasibility tolerance

c_float **eps_dual_inf**
: dual infeasibility tolerance

c_float **alpha**
: relaxation parameter

linsys_solver_type **linsys_solver**
: linear system solver to use

c_float **delta**
: regularization parameter for polish

c_int **polish**
: boolean, polish ADMM solution

c_int **polish_refine_iter**
: iterative refinement steps in polish

c_int **verbose**
: boolean, write out progres

c_int **scaled_termination**
: boolean, use scaled termination criteria

c_int **check_termination**
: integer, check termination interval. If 0, termination checking is disabled

c_int **warm_start**
: boolean, warm start

Solution

struct OSQPSolution
: Solution structure

Public Members

c_float ***x**
: Primal solution.

c_float *__y__
: Lagrange multiplier associated to $l <= Ax <= u$.

Info

__struct OSQPInfo__
: Solver return nformation

Public Members

c_int __iter__
: number of iterations taken

char __status__[32]
: status string, e.g. 'solved'

c_int __status_val__
: status as c_int, defined in constants.h

c_int __status_polish__
: polish status: successful (1), unperformed (0), (-1) unsuccessful

c_float __obj_val__
: primal objective

c_float __pri_res__
: norm of primal residual

c_float __dua_res__
: norm of dual residual

c_float __setup_time__
: time taken for setup phase (seconds)

c_float __solve_time__
: time taken for solve phase (seconds)

c_float __polish_time__
: time taken for polish phase (seconds)

c_float __run_time__
: total time (seconds)

c_int __rho_updates__
: number of rho updates

c_float __rho_estimate__
: best rho estimate so far from residuals

Workspace

__struct OSQPWorkspace__
: OSQP Workspace

Vector used to store a vectorized rho parameter

c_float ***rho_vec**
vector of rho values

c_float ***rho_inv_vec**
vector of inv rho values

Iterates

c_float ***x**
Iterate x.

c_float ***y**
Iterate y.

c_float ***z**
Iterate z.

c_float ***xz_tilde**
Iterate xz_tilde.

c_float ***x_prev**
Previous x.

NB: Used also as workspace vector for dual residual

c_float ***z_prev**
Previous z.

NB: Used also as workspace vector for primal residual

Primal and dual residuals workspace variables

Needed for residuals computation, tolerances computation, approximate tolerances computation and adapting rho

c_float ***Ax**
Scaled A * x.

c_float ***Px**
Scaled P * x.

c_float ***Aty**
Scaled A * x.

Primal infeasibility variables

c_float ***delta_y**
Difference of consecutive dual iterates.

c_float ***Atdelta_y**
A' * delta_y.

Dual infeasibility variables

c_float ***delta_x**
Difference of consecutive primal iterates.

c_float ***Pdelta_x**
P * delta_x.

c_float ***Adelta_x**
A * delta_x.

Temporary vectors used in scaling

c_float ***D_temp**
temporary primal variable scaling vectors

c_float ***D_temp_A**
temporary primal variable scaling vectors storing norms of A columns

c_float ***E_temp**
temporary constraints scaling vectors storing norms of A' columns

Public Members

OSQPData ***data**
Problem data to work on (possibly scaled)

LinSysSolver ***linsys_solver**
Linear System solver structure.

OSQPPolish ***pol**
Polish structure.

c_int ***constr_type**
Type of constraints: loose (-1), equality (1), inequality (0)

OSQPSettings ***settings**
Problem settings.

OSQPScaling ***scaling**
Scaling vectors.

OSQPSolution ***solution**
Problem solution.

OSQPInfo ***info**
Solver information.

OSQPTimer ***timer**
Timer object.

c_int **first_run**
flag indicating whether the solve function has been run before

c_int **summary_printed**
Has last summary been printed? (true/false)

Scaling

struct OSQPScaling
Problem scaling matrices stored as vectors

Public Members

c_float **c**
cost function scaling

c_float ***D**
primal variable scaling

c_float ***E**
dual variable scaling

c_float **cinv**
cost function rescaling

c_float ***Dinv**
primal variable rescaling

c_float ***Einv**
dual variable rescaling

Polish

struct OSQPPolish
Polish structure

Public Members

csc ***Ared**
Active rows of A. Ared = vstack[Alow, Aupp]

c_int **n_low**
number of lower-active rows

c_int **n_upp**
number of upper-active rows

c_int ***A_to_Alow**
Maps indices in A to indices in Alow.

c_int ***A_to_Aupp**
Maps indices in A to indices in Aupp.

c_int ***Alow_to_A**
Maps indices in Alow to indices in A.

c_int ***Aupp_to_A**
Maps indices in Aupp to indices in A.

c_float ***x**
optimal x-solution obtained by polish

c_float ***z**
optimal z-solution obtained by polish

c_float *__y__
: optimal y-solution obtained by polish

c_float __obj_val__
: objective value at polished solution

c_float __pri_res__
: primal residual at polished solution

c_float __dua_res__
: dual residual at polished solution

3.2 Python

3.2.1 Import

The OSQP module is can be imported with

```
import osqp
```

3.2.2 Setup

The solver is initialized by creating an OSQP object

```
m = osqp.OSQP()
```

The problem is specified in the setup phase by running

```
m.setup(P=P, q=q, A=A, l=l, u=u, **settings)
```

The arguments `q`, `l` and `u` are numpy arrays. The elements of `l` and `u` can be $\pm\infty$ (using `numpy.inf`).

The arguments `P` and `A` are scipy sparse matrices in CSC format. If they are sparse matrices are in another format, the interface will attemp to convert them. There is no need to specify all the arguments.

The keyword arguments `**settings` specify the solver settings. The allowed parameters are defined in *Solver settings*.

3.2.3 Solve

The problem can be solved by

```
results = m.solve()
```

The `results` object contains the primal solution `x`, the dual solution `y`, certificate of primal infeasibility `prim_inf_cert`, certificate of dual infeasibility `dual_inf_cert` and the `info` object containing the solver statistics defined in the following table

Member	Description
`iter`	Number of iterations
`status`	Solver status
`status_val`	Solver status value as in *Status values*
`status_polish`	Polishing status
`obj_val`	Objective value
`pri_res`	Primal residual
`dua_res`	Dual residual
`setup_time`	Setup time
`solve_time`	Solve time
`polish_time`	Polish time
`run_time`	Total run time: setup + solve + polish
`rho_estimate`	Optimal rho estimate
`rho_updates`	Number of rho updates

Note that if multiple solves are executed from single setup, then after the first one `run_time` includes only `solve_time` + `polish_time`.

3.2.4 Update

Part of problem data and settings can be updated without requiring a new problem setup.

Update problem vectors

Vectors `q`, `l` and `u` can be updated with new values `q_new`, `l_new` and `u_new` by just running

```
m.update(q=q_new, l=l_new, u=u_new)
```

The user does not have to specify all the keyword arguments.

Update settings

Settings can be updated by running

```
m.update_settings(**kwargs)
```

where `kwargs` are the settings that can be updated which are marked with an * in *Solver settings*.

3.2.5 Warm start

OSQP automatically warm starts primal and dual variables from the previous QP solution. If you would like to warm start their values manually, you can use

```
m.warm_start(x=x0, y=y0)
```

where `x0` and `y0` are the new primal and dual variables.

3.3 Julia

3.3.1 Load the module

The OSQP module is can be load with

```
using OSQP
```

3.3.2 Setup

The solver is initialized by creating an OSQP Model

```
m = OSQP.Model()
```

The problem is specified in the setup phase by running

```
OSQP.setup!(m; P=P, q=q, A=A, l=l, u=u, settings...)
```

The arguments `q`, `l` and `u` are `Vector{Float64}`. The elements of `l` and `u` can be $\pm\infty$ (using `Inf`).

The arguments `P` and `A` are sparse matrices of type `SparseMatrixCSC`. If they are sparse matrices are in another format, the interface will attemp to convert them. There is no need to specify all the arguments.

The argument `settings` specifies the solver settings. Settings can also be passed as indipendent keyword arguments such as `max_iter=1000`. The allowed parameters are defined in *Solver settings*.

3.3.3 Solve

The problem can be solved by

```
results = OSQP.solve!(m)
```

The output `results` contains the primal solution `x`, the dual solution `y`, certificate of primal infeasibility `prim_inf_cert`, certificate of dual infeasibility `dual_inf_cert` and the `info` object containing the solver statistics defined in the following table

Member	Description
`iter`	Number of iterations
`status`	Solver status
`status_val`	Solver status value as in *Status values*
`status_polish`	Polishing status
`obj_val`	Objective value
`pri_res`	Primal residual
`dua_res`	Dual residual
`setup_time`	Setup time
`solve_time`	Solve time
`polish_time`	Polish time
`run_time`	Total run time: setup + solve + polish
`rho_estimate`	Optimal rho estimate
`rho_updates`	Number of rho updates

Note that if multiple solves are executed from single setup, then after the first one `run_time` includes only `solve_time` + `polish_time`.

3.3.4 Update

Part of problem data and settings can be updated without requiring a new problem setup.

Update problem vectors

Vectors `q`, `l` and `u` can be updated with new values `q_new`, `l_new` and `u_new` by just running

```
OSQP.update!(m; q=q_new, l=l_new, u=u_new)
```

The user does not have to specify all the keyword arguments.

Update settings

Settings can be updated by running

```
OSQP.update_settings!(m; new_settings)
```

where `new_settings` are the new settings specified as keyword arguments that can be updated which are marked with an * in *Solver settings*.

3.3.5 Warm start

OSQP automatically warm starts primal and dual variables from the previous QP solution. If you would like to warm start their values manually, you can use

```
OSQP.warm_start!(m; x=x0, y=y0)
```

where `x0` and `y0` are the new primal and dual variables.

3.4 Matlab

3.4.1 Setup

The solver is initialized by creating an OSQP object

```
m = osqp;
```

The problem is specified in the setup phase by running

```
m.setup(P, q, A, l, u, varargin)
```

The arguments `q`, `l` and `u` are arrays. The elements of `l` and `u` can be $\pm\infty$ (using `Inf`). The arguments `P` and `A` are sparse matrices.

There is no need to specify all the problem data. They can be omitted by writing `[]`.

The last argument `varargin` specifies the solver options. You can pass the options in two ways. You can either set the individual parameters as field-value pairs, e.g.,

```
m.setup(P, q, A, l, u, 'eps_abs', 1e-04, 'eps_rel', 1e-04);
```

Alternatively, you can create a structure containing all the settings, change some of the fields and then pass it as the last argument

```
settings = m.default_settings();
settings.eps_abs = 1e-04;
settings.eps_rel = 1e-04;
m.setup(P, q, A, l, u, settings);
```

The allowed settings are defined in *Solver settings*.

3.4.2 Solve

The problem can be solved by

```
results = m.solve();
```

The `results` structure contains the primal solution `x`, the dual solution `y`, certificate of primal infeasibility `prim_inf_cert`, certificate of dual infeasibility `dual_inf_cert` and the `info` structure containing the solver statistics defined in the following table

Member	Description
`iter`	Number of iterations
`status`	Solver status
`status_val`	Solver status value as in *Status values*
`status_polish`	Polishing status
`obj_val`	Objective value
`pri_res`	Primal residual
`dua_res`	Dual residual
`setup_time`	Setup time
`solve_time`	Solve time
`polish_time`	Polish time
`run_time`	Total run time: setup + solve + polish
`rho_estimate`	Optimal rho estimate
`rho_updates`	Number of rho updates

Note that if multiple solves are executed from single setup, then after the first one `run_time` includes only `solve_time` + `polish_time`.

3.4.3 Update

Part of problem data and settings can be updated without requiring a new problem setup.

Update problem data

Vectors `q`, `l` and `u` can be updated with new values `q_new`, `l_new` and `u_new` by just running

```
m.update('q', q_new, 'l', l_new, 'u', u_new);
```

The user does not have to specify all the arguments.

Update settings

Settings can be updated by running

```
m.update_settings(varargin);
```

where `varargin` argument is described in *Setup*. The allowed settings that can be updated are marked with an * in *Solver settings*.

3.4.4 Warm start

OSQP automatically warm starts primal and dual variables from the previous QP solution. If you would like to warm start their values manually, you can use

```
m.warm_start('x', x0, 'y', y0)
```

where `x0` and `y0` are the new primal and dual variables.

3.5 Solver settings

The solver settings are displayed in the following table. The settings marked with * can be changed without running the setup method again.

Argument	Description	Allowed values	Default value
`rho` *	ADMM rho step	0 < `rho`	0.1
`sigma`	ADMM sigma step	0 < `sigma`	1e-06
`max_iter` *	Maximum number of iterations	0 < `max_iter` (integer)	2500
`eps_abs` *	Absolute tolerance	0 < `eps_abs`	1e-03
`eps_rel` *	Relative tolerance	0 < `eps_rel`	1e-03
`eps_prim_inf` *	Primal infeasibility tolerance	0 < `eps_prim_inf`	1e-04
`eps_dual_inf` *	Dual infeasibility tolerance	0 < `eps_dual_inf`	1e-04
`alpha` *	ADMM overrelaxation parameter	0 < `alpha` < 2	1.6
`linsys_solver`	Linear systems solver type	See *Linear Systems Solvers*	suitesparse ldl
`delta` *	Polishing regularization parameter	0 < `delta`	1e-06
`polish` *	Perform polishing	True/False	True
`polish_refine_iter` *	Refinement iterations in polish	0 < `polish_refine_iter` (integer)	3
`verbose` *	Print output	True/False	True
`scaled_termination` *	Scaled termination conditions	True/False	False
`check_termination` *	Check termination interval	0 (disabled) or 0 < `check_termination` (integer)	True
`warm_start` *	Perform warm starting	True/False	True
`scaling`	Number of scaling iterations	0 (disabled) or 0 < `scaling` (integer)	10
`adaptive_rho`	Adaptive rho	True/False	True
`adaptive_rho_interval`	Adaptive rho interval	0 (automatic) or 0 < `adaptive_rho_interval` (integer)	0
`adaptive_rho_tolerance`	Tolerance for adapting rho	1 <= `adaptive_rho_tolerance`	5
`adaptive_rho_fraction`	Adaptive rho interval as fraction of setup time (auto mode)	0 < `adaptive_rho_fraction`	0.4

The boolean values `True/False` are defined as `1/0` in the C/C++ interfaces.

3.6 Linear Systems Solvers

The settings parameter `linsys_solver` defines the solver for the linear system. In C/C++ it corresponds to an integer `c_int` (see *Data types*) and in the other high level languages to a string.

Solver	String option	C/C++ Constant
SuiteSparse LDL	"suitesparse ldl"	`SUITESPARSE_LDL_SOLVER`
MKL Pardiso	"mkl pardiso"	`MKL_PARDISO_SOLVER`

To add new linear system solvers see *Interfacing new linear system solvers*.

3.7 Status values

These are the exit statuses, their respective constants and values returned by the solver as defined in constants.h. The *inaccurate* statuses define when the optimality, primal infeasibility or dual infeasibility conditions are satisfied with tolerances 10 times larger than the ones set.

Status	Constant	Value
solved	OSQP_SOLVED	1
solved inaccurate	OSQP_SOLVED_INACCURATE	2
maximum iterations reached	OSQP_MAX_ITER_REACHED	-2
primal infeasible	OSQP_PRIMAL_INFEASIBLE	-3
primal infeasible inaccurate	OSQP_PRIMAL_INFEASIBLE_INACCURATE	3
dual infeasible	OSQP_DUAL_INFEASIBLE	-4
dual infeasible inaccurate	OSQP_DUAL_INFEASIBLE_INACCURATE	4
interrupted by user	OSQP_SIGINT	-5
unsolved	OSQP_UNSOLVED	-10

CHAPTER 4

Parsers

4.1 YALMIP

YALMIP support the OSQP solver. You can easily define problems in high-level format and then specify OSQP by simply setting

```
options = sdpsettings('solver','osqp', 'max_iter', 2000);
```

where we set the `max_iter` option to `2000`.

4.2 CVXPY

CVXPY 1.0 supports the OSQP solver by default. After defining your problem, you can solve it with OSQP by just calling

```
problem.solve()
```

OSQP is the default QP solver used. To specify it explicitly together with some options you can execute

```
problem.solve(solver=OSQP, max_iter=2000)
```

where we set the `max_iter` option to `2000`.

CHAPTER 5

Code generation

OSQP can generate tailored C code that compiles into a fast and reliable solver for the given family of QP problems in which the problem data, but not its dimensions, change between problem instances.

The generated code is:

Malloc-free It does not perform any dynamic memory allocation.

Library-free It is not linked to any external library.

Division-free There are no division required in the ADMM algorithm

We make a distinction between two cases depending on which of the data are to be treated as parameters.

Vectors as parameters Vectors q, l and u can change between problem instances. This corresponds to the compiler flag `EMBEDDED=1`. ρ adaptation is not enabled. The generated code is division-free and has a very small footprint.

Matrices as parameters Both vectors q, l, u and values in matrices P and A can change between problem instances. This corresponds to the compiler flag `EMBEDDED=2`. ρ adaptation is enabled but the frequency does not rely on any timing. We assume that the sparsity patterns of P and A are fixed.

5.1 Matlab

Before generating code for a parametric problem, the problem should be first specified in the setup phase. See *Setup* for more details.

5.1.1 Codegen

The code is generated by running

```
m.codegen(dir_name, varargin)
```

The argument `dir_name` is the name of a directory where the generated code is stored. The second argument `varargin` specifies additional codegen options shown in the following table

Option	Description	Allowed values

`project_type`	Build environment	`''` (default)
		`'Makefile'`
		`'MinGW Makefiles'`
		`'Unix Makefiles'`
		`'CodeBlocks'`
		`'Xcode'`

`parameters`	Problem parameters	`'vectors'` (default) `'matrices'`
`mexname`	Name of the compiled mex interface	`'emosqp'` (default) Nonempty string
`force_rewrite`	Rewrite existing directory	`false` (default) `true`

You can pass the options as field-value pairs, e.g.,

```
m.codegen('code', 'parameters', 'matrices', 'mexname', 'emosqp');
```

If the `project_type` argument is not passed or is set to `''`, then no build files are generated.

5.1.2 Mex interface

Once the code is generated the following functions are provided through its mex interface. Each function is called as

```
emosqp('function_name');
```

where `emosqp` is the name of the mex interface specified in the previous section

emosqp (*'solve'*)
Solve the problem.

Returns

multiple variables [x, y, status_val, iter, run_time]

- **x** (*ndarray*) - Primal solution
- **y** (*ndarray*) - Dual solution
- **status_val** (*int*) - Status value as in *Status values*
- **iter** (*int*) - Number of iterations
- **run_time** (*double*) - Run time

emosqp(*'update_lin_cost'*, *q_new*)
Update linear cost.

Parameters **q_new** (*ndarray*) – New linear cost vector

emosqp(*'update_lower_bound'*, *l_new*)
Update lower bound in the constraints.

Parameters **l_new** (*ndarray*) – New lower bound vector

emosqp(*'update_upper_bound'*, *u_new*)
Update upper bound in the constraints.

Parameters **u_new** (*ndarray*) – New upper bound vector

emosqp(*'update_bounds'*, *l_new*, *u_new*)
Update lower and upper bounds in the constraints.

Parameters

- **l_new** (*ndarray*) – New lower bound vector
- **u_new** (*ndarray*) – New upper bound vector

If the code is generated with the option `parameters` set to `'matrices'`, then the following functions are also provided

emosqp(*'update_P'*, *Px*, *Px_idx*, *Px_n*)
Update nonzero entries of the quadratic cost matrix.

Parameters

- **Px** (*ndarray*) – Values of entries to be updated
- **Px_idx** (*ndarray*) – Indices of entries to be updated. Pass `[]` if all the indices are to be updated
- **Px_n** (*int*) – Number of entries to be updated. Used only if Px_idx is not `[]`.

emosqp(*'update_A'*, *Ax*, *Ax_idx*, *Ax_n*)
Update nonzero entries of the constraint matrix.

Parameters

- **Ax** (*ndarray*) – Values of entries to be updated
- **Ax_idx** (*ndarray*) – Indices of entries to be updated. Pass `[]` if all the indices are to be updated
- **Ax_n** (*int*) – Number of entries to be updated. Used only if Ax_idx is not `[]`.

emosqp(*'update_P_A'*, *Px*, *Px_idx*, *Px_n*, *Ax*, *Ax_idx*, *Ax_n*)
Update nonzero entries of the quadratic cost and constraint matrices.

Parameters

- **Px** (*ndarray*) – Values of entries to be updated
- **Px_idx** (*ndarray*) – Indices of entries to be updated. Pass `[]` if all the indices are to be updated
- **Px_n** (*int*) – Number of entries to be updated. Used only if Px_idx is not `[]`.
- **Ax** (*ndarray*) – Values of entries to be updated
- **Ax_idx** (*ndarray*) – Indices of entries to be updated. Pass `[]` if all the indices are to be updated
- **Ax_n** (*int*) – Number of entries to be updated. Used only if Ax_idx is not `[]`.

You can update all the nonzero entries in matrix A by running

```
emosqp('update_A', Ax_new, [], 0);
```

See C/C++ *Sublevel API* for more details on the input arguments.

5.2 Python

Before generating code for a parametric problem, the problem should be first specified in the setup phase. See *Setup* for more details.

5.2.1 Codegen

The code is generated by running

```
m.codegen(dir_name, **opts)
```

The argument `dir_name` is the name of a directory where the generated code is stored. Additional codegen options are shown in the following table

Option	Description	Allowed values

`project_type`	Build environment	`''` (default)
		`'Makefile'`
		`'MinGW Makefiles'`
		`'Unix Makefiles'`
		`'CodeBlocks'`
		`'Xcode'`

`parameters`	Problem parameters	`'vectors'` (default) `'matrices'`
`python_ext_name`	Name of the generated Python module	`'emosqp'` (default) Nonempty string
`force_rewrite`	Rewrite existing directory	`False` (default) `True`

The options are passed using named arguments, e.g.,

```
m.codegen('code', parameters='matrices', python_ext_name='emosqp')
```

If the `project_type` argument is not passed or is set to `''`, then no build files are generated.

5.2.2 Extension module API

Once the code is generated, you can import a light python wrapper with

```
import emosqp
```

where `emosqp` is the extension name given in the previous section. The module imports the following functions

solve()
Solve the problem.

Returns

tuple (x, y, status_val, iter, run_time)

- **x** (*ndarray*) - Primal solution
- **y** (*ndarray*) - Dual solution
- **status_val** (*int*) - Status value as in *Status values*
- **iter** (*int*) - Number of iterations
- **run_time** (*double*) - Run time

update_lin_cost(*q_new*)
Update linear cost.

Parameters **q_new** (*ndarray*) – New linear cost vector

update_lower_bound(*l_new*)
Update lower bound in the constraints.

Parameters **l_new** (*ndarray*) – New lower bound vector

update_upper_bound(*u_new*)
Update upper bound in the constraints.

Parameters **u_new** (*ndarray*) – New upper bound vector

update_bounds(*l_new*, *u_new*)
Update lower and upper bounds in the constraints.

Parameters

- **l_new** (*ndarray*) – New lower bound vector
- **u_new** (*ndarray*) – New upper bound vector

If the code is generated with the option `parameters` set to `'matrices'`, the following functions are also provided

update_P(*Px*, *Px_idx*, *Px_n*)
Update nonzero entries of the quadratic cost matrix.

Parameters

- **Px** (*ndarray*) – Values of entries to be updated
- **Px_idx** (*ndarray*) – Indices of entries to be updated. Pass `None` if all the indices are to be updated
- **Px_n** (*int*) – Number of entries to be updated. Used only if Px_idx is not `None`.

update_A(*Ax*, *Ax_idx*, *Ax_n*)
Update nonzero entries of the constraint matrix.

Parameters

- **Ax** (*ndarray*) – Values of entries to be updated

- **Ax_idx** (*ndarray*) – Indices of entries to be updated. Pass `None` if all the indices are to be updated
- **Ax_n** (*int*) – Number of entries to be updated. Used only if Ax_idx is not `None`.

update_P_A(*Px*, *Px_idx*, *Px_n*, *Ax*, *Ax_idx*, *Ax_n*)

Update nonzero entries of the quadratic cost and constraint matrices.

Parameters

- **Px** (*ndarray*) – Values of entries to be updated
- **Px_idx** (*ndarray*) – Indices of entries to be updated. Pass `None` if all the indices are to be updated
- **Px_n** (*int*) – Number of entries to be updated. Used only if Px_idx is not `None`.
- **Ax** (*ndarray*) – Values of entries to be updated
- **Ax_idx** (*ndarray*) – Indices of entries to be updated. Pass `None` if all the indices are to be updated
- **Ax_n** (*int*) – Number of entries to be updated. Used only if Ax_idx is not `None`.

You can update all the nonzero entries in matrix A by running

```
emosqp.update_A(Ax_new, None, 0);
```

See C/C++ *Sublevel API* for more details on the input arguments.

CHAPTER 6

Examples

6.1 Demo

Consider the following QP

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T \begin{bmatrix} 4 & 1 \\ 1 & 2 \end{bmatrix} x + \begin{bmatrix} 1 \\ 1 \end{bmatrix}^T x \\ \text{subject to} & \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \leq \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} x \leq \begin{bmatrix} 1 \\ 0.7 \\ 0.7 \end{bmatrix} \end{array}$$

We show below how to solve the problem in Python, Matlab and C.

6.1.1 Python

```
import osqp
import scipy.sparse as sparse
import numpy as np

# Define problem data
P = sparse.csc_matrix([[4, 1], [1, 2]])
q = np.array([1, 1])
A = sparse.csc_matrix([[1, 1], [1, 0], [0, 1]])
l = np.array([1, 0, 0])
u = np.array([1, 0.7, 0.7])

# Create an OSQP object
prob = osqp.OSQP()

# Setup workspace and change alpha parameter
prob.setup(P, q, A, l, u, alpha=1.0)
```

```
# Solve problem
res = prob.solve()
```

6.1.2 Matlab

```
% Define problem data
P = sparse([4, 1; 1, 2]);
q = [1; 1];
A = sparse([1, 1; 1, 0; 0, 1]);
l = [1; 0; 0];
u = [1; 0.7; 0.7];

% Create an OSQP object
prob = osqp;

% Setup workspace and change alpha parameter
prob.setup(P, q, A, l, u, 'alpha', 1);

% Solve problem
res = prob.solve();
```

6.1.3 Julia

```
import OSQP

# Define problem data
P = sparse([4. 1.; 1. 2.])
q = [1.; 1.]
A = sparse([1. 1.; 1. 0.; 0. 1.])
u = [1.; 0.7; 0.7]
l = [1.; 0.; 0.]

# Crate OSQP object
prob = OSQP.Model()

# Setup workspace and change alpha parameter
OSQP.setup!(prob; P=P, q=q, A=A, l=l, u=u, alpha=1)

# Solve problem
results = OSQP.solve!(prob)
```

6.1.4 C

```
#include "osqp.h"

int main(int argc, char **argv) {
    // Load problem data
    c_float P_x[4] = {4.00, 1.00, 1.00, 2.00, };
    c_int P_nnz = 4;
    c_int P_i[4] = {0, 1, 0, 1, };
    c_int P_p[3] = {0, 2, 4, };
    c_float q[2] = {1.00, 1.00, };
```

```
    c_float A_x[4] = {1.00, 1.00, 1.00, 1.00, };
    c_int A_nnz = 4;
    c_int A_i[4] = {0, 1, 0, 2, };
    c_int A_p[3] = {0, 2, 4, };
    c_float l[3] = {1.00, 0.00, 0.00, };
    c_float u[3] = {1.00, 0.70, 0.70, };
    c_int n = 2;
    c_int m = 3;

    // Problem settings
    OSQPSettings * settings = (OSQPSettings *)c_malloc(sizeof(OSQPSettings));

    // Structures
    OSQPWorkspace * work;  // Workspace
    OSQPData * data;  // OSQPData

    // Populate data
    data = (OSQPData *)c_malloc(sizeof(OSQPData));
    data->n = n;
    data->m = m;
    data->P = csc_matrix(data->n, data->n, P_nnz, P_x, P_i, P_p);
    data->q = q;
    data->A = csc_matrix(data->m, data->n, A_nnz, A_x, A_i, A_p);
    data->l = l;
    data->u = u;


    // Define Solver settings as default
    set_default_settings(settings);
    settings->alpha = 1.0; // Change alpha parameter

    // Setup workspace
    work = osqp_setup(data, settings);

    // Solve Problem
    osqp_solve(work);

    // Cleanup
    osqp_cleanup(work);
    c_free(data->A);
    c_free(data->P);
    c_free(data);
    c_free(settings);

    return 0;
};
```

6.2 Huber fitting

Huber fitting or the *robust least-squares problem* performs linear regression under the assumption that there are outliers in the data. The fitting problem is written as

$$\text{minimize} \quad \sum_{i=1}^{m} \phi_{\text{hub}}(a_i^T x - b_i),$$

with the Huber penalty function $\phi_{\text{hub}} : \mathbf{R} \to \mathbf{R}$ defined as

$$\phi_{\text{hub}}(u) = \begin{cases} u^2 & |u| \leq 1 \\ (2|u| - 1) & |u| > 1 \end{cases}$$

The problem has the following equivalent form,

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}u^T u + \mathbf{1}^T v \\ \text{subject to} & -u - v \leq Ax - b \leq u + v \\ & 0 \leq u \leq 1 \\ & v \geq 0 \end{array}$$

6.2.1 Python

```
import osqp
import numpy as np
import scipy as sp
import scipy.sparse as sparse

# Generate problem data
sp.random.seed(1)
n = 10
m = 100
Ad = sparse.random(m, n, density=0.5, format='csc')
x_true = np.random.randn(n) / np.sqrt(n)
ind95 = (np.random.rand(m) < 0.95).astype(float)
b = Ad.dot(x_true) + np.multiply(0.5*np.random.randn(m), ind95) \
    + np.multiply(10.*np.random.rand(m), 1. - ind95)

# OSQP data
Im = sparse.eye(m)
P = sparse.block_diag((sparse.csc_matrix((n, n)), Im,
                       sparse.csc_matrix((m, m)))), format='csc')
q = np.append(np.zeros(m+n), np.ones(m))
A = sparse.vstack([
        sparse.hstack([Ad, Im, Im]),
        sparse.hstack([Ad, -Im, -Im]),
        sparse.hstack([sparse.csc_matrix((2*m, n)), sparse.eye(2*m)])
        ]).tocsc()
l = np.hstack([b, -np.inf*np.ones(m), np.zeros(2*m)])
u = np.hstack([np.inf*np.ones(m), b, np.ones(m), np.inf*np.ones(m)])

# Create an OSQP object
prob = osqp.OSQP()

# Setup workspace
prob.setup(P, q, A, l, u)

# Solve problem
res = prob.solve()
```

6.2.2 Matlab

```
% Generate problem data
rng(1)
n = 10;
m = 100;
Ad = sprandn(m, n, 0.5);
x_true = randn(n, 1) / sqrt(n);
ind95 = rand(m, 1) > 0.95;
b = Ad*x_true + 10*rand(m, 1).*ind95 + 0.5*randn(m, 1).*(1-ind95);

% OSQP data
Im = speye(m);
P = blkdiag(sparse(n, n), Im, sparse(m, m));
q = [zeros(m + n, 1); ones(m, 1)];
A = [Ad, Im, Im;
     Ad, -Im, -Im;
     sparse(2*m, n), speye(2*m)];
l = [b; -inf*ones(m, 1); zeros(2*m, 1)];
u = [inf*ones(m, 1); b; ones(m, 1); inf*ones(m, 1)];

% Create an OSQP object
prob = osqp;

% Setup workspace
prob.setup(P, q, A, l, u);

% Solve problem
res = prob.solve();
```

6.3 Lasso

Lasso is a well known technique for sparse linear regression. It is obtained by adding an ℓ_1 regularization term in the objective,

$$\text{minimize} \quad \frac{1}{2}\|Ax - b\|_2^2 + \gamma\|x\|_1$$

where $x \in \mathbf{R}^n$ is the vector of parameters, $A \in \mathbf{R}^{m \times n}$ is the data matrix, and $\gamma > 0$ is the weighting parameter. The problem has the following equivalent form,

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}y^T y + \gamma \mathbf{1}^T t \\ \text{subject to} & y = Ax - b \\ & -t \leq x \leq t \end{array}$$

In order to get a good trade-off between sparsity of the solution and quality of the linear fit, we solve the problem for varying weighting parameter γ. Since γ enters only in the linear part of the objective function, we can reuse the matrix factorization and enable warm starting to reduce the computation time.

6.3.1 Python

```
import osqp
import numpy as np
import scipy as sp
import scipy.sparse as sparse

# Generate problem data
```

```
sp.random.seed(1)
n = 10
m = 1000
Ad = sparse.random(m, n, density=0.5)
x_true = np.multiply((np.random.rand(n) > 0.8).astype(float),
                     np.random.randn(n)) / np.sqrt(n)
b = Ad.dot(x_true) + 0.5*np.random.randn(m)
gammas = np.linspace(1, 10, 11)

# Auxiliary data
In = sparse.eye(n)
Im = sparse.eye(m)
On = sparse.csc_matrix((n, n))
Onm = sparse.csc_matrix((n, m))

# OSQP data
P = sparse.block_diag((On, sparse.eye(m), On), format='csc')
q = np.zeros(2*n + m)
A = sparse.vstack([sparse.hstack([Ad, -Im, Onm.T]),
                   sparse.hstack([In, Onm, -In]),
                   sparse.hstack([In, Onm, In])]).tocsc()
l = np.hstack([b, -np.inf * np.ones(n), np.zeros(n)])
u = np.hstack([b, np.zeros(n), np.inf * np.ones(n)])

# Create an OSQP object
prob = osqp.OSQP()

# Setup workspace
prob.setup(P, q, A, l, u, warm_start=True)

# Solve problem for different values of gamma parameter
for gamma in gammas:
    # Update linear cost
    q_new = np.hstack([np.zeros(n+m), gamma*np.ones(n)])
    prob.update(q=q_new)

    # Solve
    res = prob.solve()
```

6.3.2 Matlab

```
% Generate problem data
rng(1)
n = 10;
m = 1000;
Ad = sprandn(m, n, 0.5);
x_true = (randn(n, 1) > 0.8) .* randn(n, 1) / sqrt(n);
b = Ad * x_true + 0.5 * randn(m, 1);
gammas = linspace(1, 10, 11);

% OSQP data
P = blkdiag(sparse(n, n), speye(m), sparse(n, n));
q = zeros(2*n+m, 1);
A = [Ad, -speye(m), sparse(m,n);
    speye(n), sparse(n, m), -speye(n);
    speye(n), sparse(n, m), speye(n);];
```

```
l = [b; -inf*ones(n, 1); zeros(n, 1)];
u = [b; zeros(n, 1); inf*ones(n, 1)];

% Create an OSQP object
prob = osqp;

% Setup workspace
prob.setup(P, q, A, l, u, 'warm_start', true);

% Solve problem for different values of gamma parameter
for i = 1 : length(gammas)
    % Update linear cost
    gamma = gammas(i);
    q_new = [zeros(n+m,1); gamma*ones(n,1)];
    prob.update('q', q_new);

    % Solve
    res = prob.solve();
end
```

6.3.3 YALMIP

```
% Generate problem data
rng(1)
n = 10;
m = 1000;
A = sprandn(m, n, 0.5);
x_true = (randn(n, 1) > 0.8) .* randn(n, 1) / sqrt(n);
b = A * x_true + 0.5 * randn(m, 1);
gammas = linspace(1, 10, 11);

% Define problem
x = sdpvar(n, 1);
gamma = sdpvar;
objective = 0.5*norm(A*x - b)^2 + gamma*norm(x,1);

% Solve with OSQP
options = sdpsettings('solver','osqp');
x_opt = optimizer([], objective, options, gamma, x);

% Solve problem for different values of gamma parameter
for i = 1 : length(gammas)
    x_opt(gammas(i));
end
```

6.4 Least-squares

Consider the following constrained least-squares problem

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|Ax - b\|_2^2 \\ \text{subject to} & 0 \leq x \leq 1 \end{array}$$

The problem has the following equivalent form

$$\begin{array}{ll} \text{minimize} & \frac{1}{2} y^T y \\ \text{subject to} & y = Ax - b \\ & 0 \leq x \leq 1 \end{array}$$

6.4.1 Python

```
import osqp
import numpy as np
import scipy as sp
import scipy.sparse as sparse

# Generate problem data
sp.random.seed(1)
m = 30
n = 20
Ad = sparse.random(m, n, density=0.7, format='csc')
b = np.random.randn(m)

# OSQP data
P = sparse.block_diag((sparse.csc_matrix((n, n)), sparse.eye(m)), format='csc')
q = np.zeros(n+m);
A = sparse.vstack([
        sparse.hstack([Ad, -sparse.eye(m)]),
        sparse.hstack((sparse.eye(n), sparse.csc_matrix((n, m))))
    ]).tocsc()
l = np.hstack([b, np.zeros(n)])
u = np.hstack([b, np.ones(n)])

# Create an OSQP object
prob = osqp.OSQP()

# Setup workspace
prob.setup(P, q, A, l, u)

# Solve problem
res = prob.solve()
```

6.4.2 Matlab

```
% Generate problem data
rng(1)
m = 30;
n = 20;
Ad = sprandn(m, n, 0.7);
b = randn(m, 1);

% OSQP data
P = blkdiag(sparse(n, n), speye(m));
q = zeros(n+m, 1);
A = [Ad, -speye(m);
     speye(n), sparse(n, m)];
l = [b; zeros(n, 1)];
u = [b; ones(n, 1)];
```

```
% Create an OSQP object
prob = osqp;

% Setup workspace
prob.setup(P, q, A, l, u);

% Solve problem
res = prob.solve();
```

6.4.3 YALMIP

```
% Generate data
rng(1)
m = 30;
n = 20;
A = sprandn(m, n, 0.7);
b = randn(m, 1);

% Define problem
x = sdpvar(n, 1);
objective = 0.5*norm(A*x - b)^2;
constraints = [ 0 <= x <= 1];

% Solve with OSQP
options = sdpsettings('solver','osqp');
optimize(constraints, objective, options);

% Get optimal primal and dual solution
x_opt = value(x);
y_opt = dual(constraints(1));
```

6.5 Model predictive control (MPC)

We consider the problem of controlling a linear time-invariant dinamical system to some reference state $x_r \in \mathbf{R}^{n_x}$. To achieve this we use *constrained linear-quadratic MPC*, which solves at each time step the following finite-horizon optimal control problem

$$\begin{array}{ll} \text{minimize} & (x_N - x_r)^T Q_N (x_N - x_r) + \sum_{k=0}^{N-1} (x_k - x_r)^T Q (x_k - x_r) + u_k^T R u_k \\ \text{subject to} & x_{k+1} = A x_k + B u_k \\ & x_{\min} \leq x_k \leq x_{\max} \\ & u_{\min} \leq u_k \leq u_{\max} \\ & x_0 = \bar{x} \end{array}$$

The states $x_k \in \mathbf{R}^{n_x}$ and the inputs $u_k \in \mathbf{R}^{n_u}$ are constrained to be between some lower and upper bounds. The problem is solved repeatedly for varying initial state $\bar{x} \in \mathbf{R}^{n_x}$.

6.5.1 Python

```
import osqp
import numpy as np
import scipy as sp
```

```
import scipy.sparse as sparse

# Discrete time model of a quadcopter
Ad = sparse.csc_matrix([
  [1.,      0.,     0., 0., 0., 0., 0.1,     0.,     0.,  0.,     0.,     0.    ],
  [0.,      1.,     0., 0., 0., 0., 0.,      0.1,    0.,  0.,     0.,     0.    ],
  [0.,      0.,     1., 0., 0., 0., 0.,      0.,     0.1, 0.,     0.,     0.    ],
  [0.0488,  0.,     0., 1., 0., 0., 0.0016,  0.,     0.,  0.0992, 0.,     0.    ],
  [0.,     -0.0488, 0., 0., 1., 0., 0.,     -0.0016, 0.,  0.,     0.0992, 0.    ],
  [0.,      0.,     0., 0., 0., 1., 0.,      0.,     0.,  0.,     0.,     0.0992],
  [0.,      0.,     0., 0., 0., 0., 1.,      0.,     0.,  0.,     0.,     0.    ],
  [0.,      0.,     0., 0., 0., 0., 0.,      1.,     0.,  0.,     0.,     0.    ],
  [0.,      0.,     0., 0., 0., 0., 0.,      0.,     1.,  0.,     0.,     0.    ],
  [0.9734,  0.,     0., 0., 0., 0., 0.0488,  0.,     0.,  0.9846, 0.,     0.    ],
  [0.,     -0.9734, 0., 0., 0., 0., 0.,     -0.0488, 0.,  0.,     0.9846, 0.    ],
  [0.,      0.,     0., 0., 0., 0., 0.,      0.,     0.,  0.,     0.,     0.9846]
])
Bd = sparse.csc_matrix([
  [0.,      -0.0726,  0.,     0.0726],
  [-0.0726,  0.,      0.0726, 0.    ],
  [-0.0152,  0.0152, -0.0152, 0.0152],
  [-0.,     -0.0006, -0.,     0.0006],
  [0.0006,   0.,     -0.0006, 0.0000],
  [0.0106,   0.0106,  0.0106, 0.0106],
  [0,       -1.4512,  0.,     1.4512],
  [-1.4512,  0.,      1.4512, 0.    ],
  [-0.3049,  0.3049, -0.3049, 0.3049],
  [-0.,     -0.0236,  0.,     0.0236],
  [0.0236,   0.,     -0.0236, 0.    ],
  [0.2107,   0.2107,  0.2107, 0.2107]])
[nx, nu] = Bd.shape

# Constraints
u0 = 10.5916
umin = np.array([9.6, 9.6, 9.6, 9.6]) - u0
umax = np.array([13., 13., 13., 13.]) - u0
xmin = np.array([-np.pi/6,-np.pi/6,-np.inf,-np.inf,-np.inf,-1.,
                 -np.inf,-np.inf,-np.inf,-np.inf,-np.inf,-np.inf])
xmax = np.array([ np.pi/6, np.pi/6, np.inf, np.inf, np.inf, np.inf,
                  np.inf, np.inf, np.inf, np.inf, np.inf, np.inf])

# Objective function
Q = sparse.diags([0., 0., 10., 10., 10., 10., 0., 0., 0., 5., 5., 5.])
QN = Q
R = 0.1*sparse.eye(4)

# Initial and reference states
x0 = np.zeros(12)
xr = np.array([0.,0.,1.,0.,0.,0.,0.,0.,0.,0.,0.,0.])

# Prediction horizon
N = 10

# Cast MPC problem to a QP: x = (x(0),x(1),...,x(N),u(0),...,u(N-1))
# - quadratic objective
P = sparse.block_diag([sparse.kron(sparse.eye(N), Q), QN,
                       sparse.kron(sparse.eye(N), R)])
# - linear objective
```

```
q = np.hstack([np.kron(np.ones(N), -Q.dot(xr)), -QN.dot(xr),
               np.zeros(N*nu)])
# - linear dynamics
Ax = sparse.kron(sparse.eye(N+1),-sparse.eye(nx)) + sparse.kron(sparse.eye(N+1, k=-1), Ad)
Bu = sparse.kron(sparse.vstack([sparse.csc_matrix((1, N)), sparse.eye(N)]), Bd)
Aeq = sparse.hstack([Ax, Bu])
leq = np.hstack([-x0, np.zeros(N*nx)])
ueq = leq
# - input and state constraints
Aineq = sparse.eye((N+1)*nx + N*nu)
lineq = np.hstack([np.kron(np.ones(N+1), xmin), np.kron(np.ones(N), umin)])
uineq = np.hstack([np.kron(np.ones(N+1), xmax), np.kron(np.ones(N), umax)])
# - OSQP constraints
A = sparse.vstack([Aeq, Aineq])
l = np.hstack([leq, lineq])
u = np.hstack([ueq, uineq])

# Create an OSQP object
prob = osqp.OSQP()

# Setup workspace
prob.setup(P, q, A, l, u, warm_start=True)

# Simulate in closed loop
nsim = 15
for i in range(nsim):
    # Solve
    res = prob.solve()

    # Check solver status
    if res.info.status != 'solved':
        raise ValueError('OSQP did not solve the problem!')

    # Apply first control input to the plant
    ctrl = res.x[-N*nu:-(N-1)*nu]
    x0 = Ad.dot(x0) + Bd.dot(ctrl)

    # Update initial state
    l[:nx] = -x0
    u[:nx] = -x0
    prob.update(l=l, u=u)
```

6.5.2 Matlab

```
% Discrete time model of a quadcopter
Ad = [1       0       0   0   0   0   0.1     0       0    0       0       0;
      0       1       0   0   0   0   0       0.1     0    0       0       0;
      0       0       1   0   0   0   0       0       0.1  0       0       0;
      0.0488  0       0   1   0   0   0.0016  0       0    0.0992  0       0;
      0      -0.0488  0   0   1   0   0      -0.0016  0    0       0.0992  0;
      0       0       0   0   0   1   0       0       0    0       0       0.0992;
      0       0       0   0   0   0   1       0       0    0       0       0;
      0       0       0   0   0   0   0       1       0    0       0       0;
      0       0       0   0   0   0   0       0       1    0       0       0;
      0.9734  0       0   0   0   0   0.0488  0       0    0.9846  0       0;
      0      -0.9734  0   0   0   0   0      -0.0488  0    0       0.9846  0;
```

```
      0       0       0   0   0   0   0       0       0    0       0       0.9846];
Bd = [0      -0.0726  0       0.0726;
     -0.0726  0       0.0726  0;
     -0.0152  0.0152 -0.0152  0.0152;
      0      -0.0006 -0.0000  0.0006;
      0.0006  0      -0.0006  0;
      0.0106  0.0106  0.0106  0.0106;
      0      -1.4512  0       1.4512;
     -1.4512  0       1.4512  0;
     -0.3049  0.3049 -0.3049  0.3049;
      0      -0.0236  0       0.0236;
      0.0236  0      -0.0236  0;
      0.2107  0.2107  0.2107  0.2107];
[nx, nu] = size(Bd);

% Constraints
u0 = 10.5916;
umin = [9.6; 9.6; 9.6; 9.6] - u0;
umax = [13; 13; 13; 13] - u0;
xmin = [-pi/6; -pi/6; -Inf; -Inf; -Inf; -1; -Inf(6,1)];
xmax = [ pi/6;  pi/6;  Inf;  Inf;  Inf; Inf; Inf(6,1)];

% Objective function
Q = diag([0 0 10 10 10 10 0 0 0 5 5 5]);
QN = Q;
R = 0.1*eye(4);

% Initial and reference states
x0 = zeros(12,1);
xr = [0; 0; 1; 0; 0; 0; 0; 0; 0; 0; 0; 0];

% Prediction horizon
N = 10;

% Cast MPC problem to a QP: x = (x(0),x(1),...,x(N),u(0),...,u(N-1))
% - quadratic objective
P = blkdiag( kron(speye(N),Q), QN, kron(speye(N),R) );
% - linear objective
q = [repmat(-Q*xr, N, 1); -QN*xr; zeros(N*nu,1)];
% - linear dynamics
Ax = kron(speye(N+1), -speye(nx)) + kron(sparse(diag(ones(N,1), -1)), Ad);
Bu = kron([sparse(1, N); speye(N)], Bd);
Aeq = [Ax, Bu];
leq = [-x0; zeros(N*nx,1)];
ueq = leq;
% - input and state constraints
Aineq = speye((N+1)*nx + N*nu);
lineq = [repmat(xmin, N+1, 1); repmat(umin, N, 1)];
uineq = [repmat(xmax, N+1, 1); repmat(umax, N, 1)];
% - OSQP constraints
A = [Aeq; Aineq];
l = [leq; lineq];
u = [ueq; uineq];

% Create an OSQP object
prob = osqp;

% Setup workspace
```

```
prob.setup(P, q, A, l, u, 'warm_start', true);

% Simulate in closed loop
nsim = 15;
for i = 1 : nsim
    % Solve
    res = prob.solve();

    % Check solver status
    if ~strcmp(res.info.status, 'solved')
        error('OSQP did not solve the problem!')
    end

    % Apply first control input to the plant
    ctrl = res.x((N+1)*nx+1:(N+1)*nx+nu);
    x0 = Ad*x0 + Bd*ctrl;

    % Update initial state
    l(1:nx) = -x0;
    u(1:nx) = -x0;
    prob.update('l', l, 'u', u);
end
```

6.5.3 YAMLIP

```
% Discrete time model of a quadcopter
Ad = [1       0       0   0   0   0   0.1     0       0    0       0       0;
      0       1       0   0   0   0   0       0.1     0    0       0       0;
      0       0       1   0   0   0   0       0       0.1  0       0       0;
      0.0488  0       0   1   0   0   0.0016  0       0    0.0992  0       0;
      0      -0.0488  0   0   1   0   0      -0.0016  0    0       0.0992  0;
      0       0       0   0   0   1   0       0       0    0       0       0.0992;
      0       0       0   0   0   0   1       0       0    0       0       0;
      0       0       0   0   0   0   0       1       0    0       0       0;
      0       0       0   0   0   0   0       0       1    0       0       0;
      0.9734  0       0   0   0   0   0.0488  0       0    0.9846  0       0;
      0      -0.9734  0   0   0   0   0      -0.0488  0    0       0.9846  0;
      0       0       0   0   0   0   0       0       0    0       0       0.9846];
Bd = [0      -0.0726  0       0.0726;
     -0.0726  0       0.0726  0;
     -0.0152  0.0152 -0.0152  0.0152;
      0      -0.0006 -0.0000  0.0006;
      0.0006  0      -0.0006  0;
      0.0106  0.0106  0.0106  0.0106;
      0      -1.4512  0       1.4512;
     -1.4512  0       1.4512  0;
     -0.3049  0.3049 -0.3049  0.3049;
      0      -0.0236  0       0.0236;
      0.0236  0      -0.0236  0;
      0.2107  0.2107  0.2107  0.2107];
[nx, nu] = size(Bd);

% Constraints
u0 = 10.5916;
umin = [9.6; 9.6; 9.6; 9.6] - u0;
umax = [13; 13; 13; 13] - u0;
```

```
xmin = [-pi/6; -pi/6; -Inf; -Inf; -Inf; -1; -Inf(6,1)];
xmax = [ pi/6;  pi/6;  Inf;  Inf;  Inf; Inf; Inf(6,1)];

% Objective function
Q = diag([0 0 10 10 10 10 0 0 0 5 5 5]);
QN = Q;
R = 0.1*eye(4);

% Initial and reference states
x0 = zeros(12,1);
xr = [0; 0; 1; 0; 0; 0; 0; 0; 0; 0; 0; 0];

% Prediction horizon
N = 10;

% Define problem
u = sdpvar(repmat(nu,1,N),   repmat(1,1,N));
x = sdpvar(repmat(nx,1,N+1), repmat(1,1,N+1));
constraints = [xmin <= x{1} <= xmax];
objective = 0;
for k = 1 : N
    objective = objective + (x{k}-xr)'*Q*(x{k}-xr) + u{k}'*R*u{k};
    constraints = [constraints, x{k+1} == Ad*x{k} + Bd*u{k}];
    constraints = [constraints, umin <= u{k}<= umax, xmin <= x{k+1} <= xmax];
end
objective = objective + (x{N+1}-xr)'*Q*(x{N+1}-xr);
options = sdpsettings('solver','osqp');
controller = optimizer(constraints, objective,options,x{1},[u{:}]);

nsim = 15;
for i = 1 : 15
    U = controller{x0};
    x0 = Ad*x0 + Bd*U(:,1);
end
```

6.6 Portfolio optimization

Portfolio optimization seeks to allocate assets in a way that maximizes the risk adjusted return,

$$\begin{array}{ll} \text{maximize} & \mu^T x - \gamma \left(x^T \Sigma x\right) \\ \text{subject to} & \mathbf{1}^T x = 1 \\ & x \geq 0 \end{array}$$

where $x \in \mathbf{R}^n$ represents the portfolio, $\mu \in \mathbf{R}^n$ the vector of expected returns, $\gamma > 0$ the risk aversion parameter, and $\Sigma \in \mathbf{S}^n_+$ the risk model covariance matrix. The risk model is usually assumed to be the sum of a diagonal and a rank $k < n$ matrix,

$$\Sigma = FF^T + D,$$

where $F \in \mathbf{R}^{n \times k}$ is the factor loading matrix and $D \in \mathbf{S}^n_+$ is a diagonal matrix describing the asset-specific risk. The resulting problem has the following equivalent form,

$$\begin{array}{ll} \text{minimize} & \frac{1}{2} x^T D x + \frac{1}{2} y^T y - \frac{1}{2\gamma} \mu^T x \\ \text{subject to} & y = F^T x \\ & \mathbf{1}^T x = 1 \\ & x \geq 0 \end{array}$$

6.6.1 Python

```
import osqp
import numpy as np
import scipy as sp
import scipy.sparse as sparse

# Generate problem data
sp.random.seed(1)
n = 100
k = 10
F = sparse.random(n, k, density=0.7, format='csc')
D = sparse.diags(np.random.rand(n) * np.sqrt(k), format='csc')
mu = np.random.randn(n)
gamma = 1

# OSQP data
P = sparse.block_diag((D, sparse.eye(k)), format='csc')
q = np.hstack([-mu / (2*gamma), np.zeros(k)])
A = sparse.vstack([
        sparse.hstack([F.T, -sparse.eye(k)]),
        sparse.hstack([sparse.csc_matrix(np.ones((1, n))), sparse.csc_matrix((1, k))]),
        sparse.hstack((sparse.eye(n), sparse.csc_matrix((n, k))))
    ]).tocsc()
l = np.hstack([np.zeros(k), 1., np.zeros(n)])
u = np.hstack([np.zeros(k), 1., np.ones(n)])

# Create an OSQP object
prob = osqp.OSQP()

# Setup workspace
prob.setup(P, q, A, l, u)

# Solve problem
res = prob.solve()
```

6.6.2 Matlab

```
% Generate problem data
rng(1)
n = 100;
k = 10;
F = sprandn(n, k, 0.7);
D = sparse(diag( sqrt(k)*rand(n,1) ));
mu = randn(n, 1);
gamma = 1;

% OSQP data
P = blkdiag(D, speye(k));
q = [-mu/(2*gamma); zeros(k, 1)];
A = [F', -speye(k);
     ones(1, n), zeros(1, k);
     speye(n), sparse(n, k)];
l = [zeros(k, 1); 1; zeros(n, 1)];
u = [zeros(k, 1); 1; ones(n, 1)];
```

```
% Create an OSQP object
prob = osqp;

% Setup workspace
prob.setup(P, q, A, l, u);

% Solve problem
res = prob.solve();
```

6.6.3 YALMIP

```
% Generate problem data
rng(1)
n = 100;
k = 10;
F = sprandn(n, k, 0.7);
D = sparse(diag( sqrt(k)*rand(n,1) ));
mu = randn(n, 1);
gamma = 1;
Sigma = F*F' + D;

% Define problem
x = sdpvar(n, 1);
objective = gamma * (x'*Sigma*x) - mu'*x;
constraints = [sum(x) == 1, x >= 0];

% Solve with OSQP
options = sdpsettings('solver','osqp');
optimize(constraints, objective, options);

% Get optimal primal and dual solution
x_opt = value(x);
y_opt = dual(constraints(1));
```

6.7 Support vector machine (SVM)

Support vector machine seeks an affine function that approximately classifies the two sets of points. The problem can be stated as

$$\text{minimize} \quad \frac{1}{2}x^T x + \gamma \sum_{i=1}^{m} \max(0, b_i a_i^T x + 1),$$

where $b_i \in \{-1, +1\}$ is a set label, and a_i is a vector of features for the i-th point. The problem has the following equivalent form

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T x + \gamma \mathbf{1}^T t \\ \text{subject to} & t \geq \text{diag}(b)Ax + 1 \\ & t \geq 0, \end{array}$$

where $\text{diag}(b)$ denotes the diagonal matrix with elements of b on its diagonal.

6.7.1 Python

```
import osqp
import numpy as np
import scipy as sp
import scipy.sparse as sparse

# Generate problem data
sp.random.seed(1)
n = 10
m = 1000
N = int(m / 2)
gamma = 1.0
b = np.hstack([np.ones(N), -np.ones(N)])
A_upp = sparse.random(N, n, density=0.5)
A_low = sparse.random(N, n, density=0.5)
Ad = sparse.vstack([
        A_upp / np.sqrt(n) + (A_upp != 0.).astype(float) / n,
        A_low / np.sqrt(n) - (A_low != 0.).astype(float) / n
     ]).tocsc()

# OSQP data
Im = sparse.eye(m)
P = sparse.block_diag((sparse.eye(n), sparse.csc_matrix((m, m))), format='csc')
q = np.hstack([np.zeros(n), gamma*np.ones(m)])
A = sparse.vstack([
        sparse.hstack([sparse.diags(b).dot(Ad), -Im]),
        sparse.hstack([sparse.csc_matrix((m, n)), Im])
    ]).tocsc()
l = np.hstack([-np.inf*np.ones(m), np.zeros(m)])
u = np.hstack([-np.ones(m), np.inf*np.ones(m)])

# Create an OSQP object
prob = osqp.OSQP()

# Setup workspace
prob.setup(P, q, A, l, u)

# Solve problem
res = prob.solve()
```

6.7.2 Matlab

```
% Generate problem data
rng(1)
n = 10;
m = 1000;
N = ceil(m/2);
gamma = 1;
A_upp = sprandn(N, n, 0.5);
A_low = sprandn(N, n, 0.5);
Ad = [A_upp / sqrt(n) + (A_upp ~= 0) / n;
      A_low / sqrt(n) - (A_low ~= 0) / n];
b = [ones(N, 1); -ones(N,1)];

% OSQP data
P = blkdiag(2*speye(n), sparse(m, m));
q = [zeros(n,1); gamma*ones(m,1)];
```

```
A = [diag(b)*Ad, -speye(m);
     sparse(m, n), speye(m)];
l = [-inf*ones(m, 1); zeros(m, 1)];
u = [-ones(m, 1); inf*ones(m, 1)];

% Create an OSQP object
prob = osqp;

% Setup workspace
prob.setup(P, q, A, l, u);

% Solve problem
res = prob.solve();
```

CHAPTER 7

Contributing

OSQP is an open-source project open to any academic or commercial applications. Contributions are welcome as GitHub pull requests in any part of the project such as

- algorithm developments
- interfaces to other languages
- compatibility with new architectures
- linear system solvers interfaces

7.1 Interfacing new linear system solvers

OSQP is designed to be easily interfaced to new linear system solvers via dynamic library loading. To add a linear system solver interface you need to edit the `lin_sys/` directory subfolder `direct/` or `indirect/` depending on the type of solver. Create a subdirectory with your solver name with four files:

- Dynamic library loading: `mysolver_loader.c` and `mysolver_loader.h`.
- Linear system solution: `mysolver.c` and `mysolver.h`.

We suggest you to have a look at the MKL Pardiso solver interface for more details.

7.1.1 Dynamic library loading

In this part define the methods to load the shared library and obtain the functions required to solve the linear system. The main functions to be exported are `lh_load_mysolver(const char* libname)` and `lh_unload_mysolver()`. In addition, the file `mysolver_loader.c` must define static function pointers to the shared library functions to be loaded.

7.1.2 Linear system solution

In this part we define the core of the interface: **linear system solver object**. The main functions are the external method

- `init_linsys_solver_mysolver`: create the instance and perform the setup

and the internal methods of the object

- `free_linsys_solver_mysolver`: free the instance
- `solve_linsys_mysolver`: solve the linear system
- `update_matrices`: update problem matrices
- `update_rho_vec`: update ρ as a diagonal vector.

After the initializations these functions are assigned to the internal pointers so that, for an instance `s` they can be called as `s->free`, `s->solve`, `s->update_matrices` and `s->update_rho_vec`.

The linear system solver object is defined in `mysolver.h` as follows

```
typedef struct mysolver mysolver_solver;

struct mysolver {
    // Methods
    enum linsys_solver_type type; // Linear system solver defined in constants.h

    c_int (*solve)(struct mysolver * self, c_float * b, const OSQPSettings * settings);
    void (*free)(struct mysolver * self);
    c_int (*update_matrices)(struct mysolver * self, const csc *P, const csc *A, const␣
↪OSQPSettings *settings);
    c_int (*update_rho_vec)(struct mysolver * self, const c_float * rho_vec, const c_int m);

    // Attributes
    c_int nthreads; // Number of threads used (required!)

    // Internal attributes of the solver
    ...

    // Internal attributes required for matrix updates
    c_int * Pdiag_idx, Pdiag_n;  ///< index and number of diagonal elements in P
    c_int * PtoKKT, * AtoKKT;    ///< Index of elements from P and A to KKT matrix
    c_int * rhotoKKT;            ///< Index of rho places in KKT matrix
    ...

};

// Initialize mysolver solver
mysolver_solver *init_linsys_solver_mysolver(const csc * P, const csc * A, c_float sigma, c_
↪float * rho_vec, c_int polish);

// Solve linear system and store result in b
c_int solve_linsys_mysolver(mysolver_solver * s, c_float * b, const OSQPSettings * settings);

 // Update linear system solver matrices
c_int update_linsys_solver_matrices_mysolver(mysolver_solver * s,
                const csc *P, const csc *A, const OSQPSettings *settings);

// Update rho parameter in linear system solver structure
c_int update_linsys_solver_rho_vec_mysolver(mysolver_solver * s, const c_float * rho_vec, const␣
↪c_int m);

// Free linear system solver
void free_linsys_solver_mysolver(mysolver_solver * s);
```

The function details are coded in the `mysolver.c` file.

CHAPTER 8

Citing OSQP

If you use OSQP for published work, we encourage you to put a star on GitHub and cite the accompanying papers:

Main paper Main algorithm description, derivation and benchmark available in this preprint.

```
@article{osqp,
  author = {Stellato, B. and Banjac, G. and Goulart, P. and Bemporad, A. and Boyd, S.},
  title = {{OSQP}: An Operator Splitting Solver for Quadratic Programs},
  journal = {ArXiv e-prints},
  year = {2017},
  month = nov,
  adsnote = {Provided by the SAO/NASA Astrophysics Data System},
  adsurl = {http://adsabs.harvard.edu/abs/2017arXiv171108013S},
  archiveprefix = {arXiv},
  eprint = {1711.08013},
  keywords = {Mathematics - Optimization and Control},
  primaryclass = {math.OC},
}
```

Infeasibility detection Infeasibility detection proofs using ADMM (also for general conic programs) in this preprint.

```
@article{osqp-infeasibility,
  title   = {Infeasibility detection in the alternating direction method of multipliers for␣
↪convex optimization},
  author  = {Banjac, G. and Goulart, P. and Stellato, B. and Boyd, S.},
  journal = {optimization-online.org},
  year    = {2017},
  url     = {http://www.optimization-online.org/DB_HTML/2017/06/6058.html},
}
```

Code generation Code generation functionality and example in this paper.

```
@inproceedings{osqp-codegen,
  author = {Banjac, G. and Stellato, B. and Moehle, N. and Goulart, P. and Bemporad, A. and␣
↪Boyd, S.},
  title = {Embedded code generation using the {OSQP} solver},
```

```
  booktitle = {{IEEE} Conference on Decision and Control ({CDC})},
  year = {2017},
  month = dec,
}
```

Index

R

V