qpSWIFT: A Sparse Quadratic Programming Solver

Abhishek Pandala, Yanran Ding and Hae-Won Park

Contents

Introduction

This document provides an introduction to qpSWIFT [1], a Real-Time Sparse Quadratic Program Solver. qpSWIFT solves quadratic programs of the following form

$$\begin{aligned} \min_x \; & \frac{1}{2}x^T P x + c^T x \\ s.t. \; & Ax = b \\ & Gx \leq h \end{aligned}$$

Here, it is assumed that P is symmetric, positive semi-definite matrix and A is full row- rank. The solver employs Primal-Dual Interior Point method with Mehrotra Predictor corrector steps and Nesterov Todd scaling. For solving the linear system of equations, sparse LDL^T factorization is used along with approximate minimum degree (AMD) heuristic to minimize fill-in of the factorizations.

Features

qpSWIFT boasts the following features

- Written in ANSI-C
- Fully functional Quadratic Programming solver for embedded applications
- Tested on multiple target architectures
  - x86
  - x86_64
  - ARM
- Support for multiple interfaces
  - C/C++
  - Python
  - Matlab
  - Simulink

Download

qpSWIFT can be downloaded from the following link as

```
git clone https://github.com/abhishek-pandala/qpSWIFT
```

Installation

4.1 | C/C++

4.1.1 Linux

Add a link to the demo at the end of installation

To build qpSWIFT library from source on your system, type the following commands from the qpSWIFT project source directory

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release
```

To install the libraries, type

```
sudo cmake --build build -target install
            # or
cd build
sudo make install
```

Prerequisites

- cmake
- c, c++ compiler

4.1.2 macOS

Add a link to the demo at the end of installation

To build qpSWIFT library from source on your system, type the following commands from the qpSWIFT project source directory

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release
```

To install the libraries, type

```
sudo cmake --build build -target install
            # or
cd build
sudo make install
```

4.1.3 Windows

On Windows machine, type the following commands from qpSWIFT project source directory in the command prompt or windows powershell

```
cmake -S . -B build
```

To generate build files for a specific MSVC version (e.g., Visual Studio 15 compiler), use

```
cmake -S . -B build -G "Visual Studio 15 2017"
```

This creates the necessary build files to generate qpSWIFT libraries. Alternatively, you can use the cmake-gui to generate build files. To compile the libraries, type

```
cmake --build build -j8 --config Release
```

Please ensure that you have the permissions to install libraries on your system. The install folder can be set via `CMAKE_INSTALL_PREFIX`. To install the libraries,

```
cmake --build build --target install
```

4.1.4 Adding qpSWIFT to your project

To incorporate qpSWIFT into your project, add the following lines into your cmake file

```
find_package(qpSWIFT)
## To incorporate static library
target_link_libraries(<target_name>
                      PRIVATE qpSWIFT::qpSWIFT-static)
## To incorporate shared library
target_link_libraries(<target_name>
                      PRIVATE qpSWIFT::qpSWIFT-shared)
```

4.2 | Matlab

To build qpSWIFT matlab interface from source on your system, type the following commands from the qpSWIFT matlab directory be more specific to change into a matlab folder

Prerequisites

- matlab compatible c compiler

```
Swift_make('qpSWIFT_mex.c')
```

This creates a mex file depending on the configuration of the system. To use qpSWIFT in your projects copy this mex file into project working directory or add this folder to the matlab search path.

4.3 | Python

To build qpSWIFT python interface from source on your system, type the following commands from the qpSWIFT python directory in the system command line. more specific instructions to compile python interface

```
sudo python3 setup.py install
```

This builds and installs qpSWIFT module in your system. The other way to install python module is to type

```
pip install qpSWIFT
```

Prerequisites

- c compiler
- numpy
- distutils

4.4 | Simulink

In Progress ...

Demos

We use the following two quadratic programs as an example on each of the interface for the rest of the documentation

5.0.1 Problem Setup - 1

The following problem shows a standard QP with both equality and inequality constraints

$$
\begin{aligned}
\underset{x}{min}\ \frac{1}{2}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}^T \begin{bmatrix}5 & 1 & 0\\1 & 2 & 1\\0 & 1 & 4\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} + \begin{bmatrix}1\\2\\1\end{bmatrix}^T\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}\\
s.t.\ \begin{bmatrix}1 & -2 & 1\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} = 3\\
\begin{bmatrix}-4 & -4 & 0\\0 & 0 & -1\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} \leq \begin{bmatrix}-1\\-1\end{bmatrix}
\end{aligned}
$$

The solution for the above QP is $\begin{bmatrix}0.450 & -0.200 & 1.000\end{bmatrix}^T$

5.0.2 Problem Setup - 2

The following problem shows a standard QP with only inequality constraints

$$
\begin{aligned}
\underset{x}{min}\ \frac{1}{2}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}^T \begin{bmatrix}5 & 1 & 0\\1 & 2 & 1\\0 & 1 & 4\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} + \begin{bmatrix}1\\2\\1\end{bmatrix}^T\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}\\
s.t.\ \begin{bmatrix}-4 & -4 & 0\\0 & 0 & -1\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} \leq \begin{bmatrix}-1\\-1\end{bmatrix}
\end{aligned}
$$

The solution for the above QP is $\begin{bmatrix}0.833 & -0.583 & 1.000\end{bmatrix}^T$

5.1 | C/C++ Interface

5.1.1 C Interface

Demo file for C interface can be found at `demo/runqp.c`. To interface with qpSWIFT function, add the header file

```
#include "qpSWIFT.h"
```

qpSWIFT has three main functions, `QP_SETUP`, `QP_SOLVE` and `QP_CLEANUP`. Initialize the QP struct with

```
QP* myQP
```

The first function QP_SETUP has the following input, output arguments

```
qp_int n        /* number of Decision Variables */
qp_int m        /* number of inequality constraints */
qp_int p        /* number of equality constraints */
qp_int* Pjc     /* jc vector of P Matrix in CCS format */
qp_int* Pir     /* ir vector of P Matrix in CCS format */
qp_real* Ppr    /* pr vector of P Matrix in CCS format */
qp_int* Ajc     /* jc vector of A Matrix in CCS format */
qp_int* Air     /* ir vector of A Matrix in CCS format */
qp_real* Apr    /* pr vector of A Matrix in CCS format */
qp_int* Gjc     /* jc vector of G Matrix in CCS format */
qp_int* Gir     /* ir vector of G Matrix in CCS format */
qp_real* Gpr    /* pr vector of G Matrix in CCS format */
qp_real* c      /* cost function vector */
qp_real* h      /* inequality constraints vector */
qp_real* b      /* equality constraints vector */
qp_int sigma_d  /* desired sigma*/
qp_int Permut   /* permutation vector of KKT Matrix */
```

Here, qpSWIFT accepts matrices in Compressed Column Storage format. `sigma_d` is set to zero. The last argument Permut refers to the permutation matrix of the KKT matrix. This can be set to `NULL` if you want to use the inbuilt AMD routines to compute permutation matrix. More info regarding CCS format and Permutation matrices can be found in the appendix. If there are no equality constraints in the QP problem, then set the following arguments `Ajc, Air, Apr, B` to a `NULL` pointer and the number of equality constraints as `p = 0`. Settings for the QP can now be changed as

```
myQP->settings->maxit   /* Maximum number of Iterations of QP */
myQP->settings->reltol  /* Relative Tolerance */
myQP->settings->abstol  /* Absolute Tolerance */
myQP->settings->sigma   /* sigma Desired*/
myQP->settings->verbose /* Verbose Levels || 0 :: No Print */
                        /*                || >0 :: Print Everything */
```

The default for these settings can be found in `include/GlobalOptions.h`. The function QP_SOLVE performs the actual solution

```
Exit_Code = QP_SOLVE(myQP)
```

The `Exit_Code` has the exit flag of the qpSWIFT. The following flags are set for `ExitCode`

```
QP_OPTIMAL (0)  /* optimal solution found */
QP_KKTFAIL (1)  /* failure in solving LDL' factorization */
```

```
QP_MAXIT (2)    /* maximum number of iterations exceeded */
QP_FATAL (3)    /* unknown problem in solver */
```

The solution can be accessed via the pointer

```
myQP->x         /* Primal Solution of the QP */
myQP->y         /* Dual Solution of the QP */
myQP->z         /* Dual Solution of the QP */
myQP->s         /* Slack Variables of the QP */
```

The statistics of the solution can be accessed via `myQP->stats` pointer and have the following values

```
/* Time Statsitics */
tsetup          /* Setup time ; includes initialisation as well */
tsolve          /* QP solve time */
kkt_time        /* kkt matrix inversion time */
ldl_numeric     /* kkt matrix factorization time */
/* Time Statsitics */

/* Algorithmic Statistics */
IterationCount  /* iteration Count */
n_rx            /* norm of residual vector rx */
n_ry            /* norm of residual vector ry */
n_rz            /* norm of residual vector rz */
n_mu            /* complementary Slackness (s'z/m) */

alpha_p         /* primal Step Size */
alpha_d         /* dual Step Size  */
/* Algorithmic Statistics */

fval            /* function Value */
Flag            /* solver FLAG */
AMD_RESULT      /* AMD Compilation Result */
                /* >=0 means successfull  */
                /* <0 means unsuccesfull  */
                /* -3 means unused */
```

The last function to call is `QP_CLEANUP`.

```
QP_CLEANUP(myQP)
```

This function clears all the memory created by `QP_SETUP`. Please note that this function needs to be called after copying the QP solution and relevant statistics.

5.1.2 C++ Interface via Eigen

Demo file for C++ interface can be found at demo/runqpcpp.ccpp. The C++ interface is also similar to the C interface, after adding the header file

```
#include "qpSWIFT.h"
```

First the QP struct pointer is created as

```
QP* myQP;
```

The solver can be interfaced via the same three functions, same way as C interface or can be done via dense interface using QP_SETUP_dense function. The inputs arguments for QP_SETUP_dense are as follows

```
qp_int n         /* Number of Decision Variables */
qp_int m         /* Number of inequality constraints */
qp_int p         /* Number of equality constraints */
qp_real *P       /* data pointer of P Matrix */
qp_real *G       /* data pointer of G Matrix */
qp_real *A       /* data pointer of A Matrix */
qp_real *c       /* cost function vector */
qp_real *h       /* inequality constraints vector */
qp_real *b       /* equality constraints vector */
qp_int Permut    /* Permutation vector of KKT Matrix */
qp_int ordering  /* Indicates row major or column major ordering of P,A,G matrices*/
                 /* ROW_MAJOR_ORDERING - row major indexing of matrices */
                 /* COLUMN_MAJOR_ORDERING - row major indexing of matrices */
```

Note that all the matrices P, A and G must be in contiguous blocks of memory and must be in either row-major or column-major ordering. The settings, solution and stats can be obtained the same way as the C interface. The only difference is with the dense interface, QP_CLEANUP_dense needs to be called instead of QP_CLEANUP

5.2 Matlab Interface

Demo file can be found at matlab/demoqp. To use qpSWIFT for general quadratic program of the form

$$\begin{aligned} &\underset{x}{min}\ \frac{1}{2}x^T P x + c^T x \\ &s.t.\ Ax = b \\ &\quad\ \ Gx \le h \end{aligned}$$

type the following commands

```
[sol,basic_info,adv_info] = qpSWIFT(sparse(P),c,sparse(A),b,sparse(G),h)
                                    or
[sol,basic_info,adv_info] = qpSWIFT(sparse(P),c,sparse(A),b,sparse(G),h,opts)
```

For inequality only quadratic program of the form

$$\min_{x} \frac{1}{2}x^T Px + c^T x$$
$$s.t. Gx \leq h$$

the syntax is as follows

```
[sol,basic_info,adv_info] = qpSWIFT(sparse(P),c,sparse(G),h)
                          or
[sol,basic_info,adv_info] = qpSWIFT(sparse(P),c,sparse(G),h,opts)
```

The input arguments and their description is given below

```
P is a sparse matrix of dimension (n,n)
c is a dense column vector of size n
A is a sparse matrix of size (p,n); p is number of equality constraints
b is a dense column vector of size p
G is a sparse matrix of size (m,n); m is the number of inequality constraints
h is a dense column vector of size m
opts is a structure with the following fields
-> MAXITER : maximum number of iterations needed
-> ABSTOL  : absolute tolerance
-> RELTOL  : relative tolerance
-> SIGMA   : maximum centering allowed
-> VERBOSE : print levels ||  0 -- no print
                          || >0 -- print everything
-> Permut  : permutation vector obtained as
            KKT = [P A' G';
                   A 0   0;
                   G 0 -I];
            Permut = amd(KKT);
```

Note that opts is not mandatory and all fields of opts are also not mandatory. All input matrices must be sparse. The output arguments have the following description

```
sol represents the primal solution of the QP

basic_info has four fields
    -> Exit Flag : 0 : optimal solution found
                 : 1 : failure in factorising KKT matrix
                 : 2 : maximum number of iterations reached
                 : 3 : unknown problem in solver
    -> Iterations : number of iterations
    -> Setup Time : involves setting up QP; solving for initial guess
    -> Solve Time : solution time in ms
```

```
adv_info  has five fields
    -> Fval       : objective value of the QP
    -> KKT_Time   : time needed to solve the KKT system of equations
    -> LDL_Time   : time needed to perform LDL' factorization
    -> y          : dual variables corresponding to equality constraints
    -> z          : dual variables corresponding to inequality constraints
    -> s          : primal slack variables
```

5.3 | Python Interface

Demo file can be found at `python/demoqp.py`. To run qpSWIFT in your python script

```
import qpSWIFT as qp
res = qp.run(c,h,P,G,A,b,opts)
```

The last three arguments `A,b,opts` are optional. The input arguments and their description is as follows

```
P is a numpy matrix of dimension (n,n)
c is a numpy column vector of size n
A is a numpy matrix of size (p,n); p is number of equality constraints
b is a numpy column vector of size p
G is a numpy matrix of size (m,n); m is the number of inequality constraints
h is a numpy column vector of size m
opts is a dictionary with the following fields
    -> MAXITER : maximum number of iterations needed
    -> ABSTOL  : absolute tolerance
    -> RELTOL  : relative tolerance
    -> SIGMA   : maximum centering allowed
    -> VERBOSE : PRINT LEVELS ||  0 -- No Print
                              || >0 -- Print everything
    -> Permut  : permutation vector
```

opts is not mandatory and all fields of opts are also not mandatory. The output arguments and their description is as follows

```
res represents a dictionary class with the following key value pairs

*   [sol] : Basic Solution represented as numpy vector

*   [basic_info] : Dictionary class with four key-value pairs
        -> Exit Flag : 0 : Optimal Solution Found
                     : 1 : Failure in factorising KKT matrix
                     : 2 : Maximum Number of Iterations Reached
                     : 3 : Unknown Problem in Solver
        -> Iterations : Number of Iterations
```

```
        -> Setup Time : Involves setting up QP; solving for initial guess
        -> Solve Time : Solution Time in ms

*   [adv_info] : Dictionary class with five key-value pairs
    -> Fval       : Objective Value of the QP
    -> KKT_Time   : Time needed to solve the KKT system of equations
    -> LDL_Time   : Time needed to perform LDL' factorization
    -> y          : Dual Variables
    -> z          : Dual Variables
    -> s          : Primal Slack Variables
```

5.4 | Simulink Interface

In Progress ...

Acnowledgement and License

The code structure of qpSWIFT is heavily inspired from ecos [2]. The algorithm details can be found in [1]. qpSWIFT is distributed with GNU General Public License v3.0 with an aim to help researchers.

Citing qpSWIFT

If you like qpSWIFT and are using it in your work, please cite the following paper

```
@article{pandala2019qpswift,
title     = {qpSWIFT: A Real-Time Sparse Quadratic Program Solver for
             Robotic Applications},
author    = {Pandala, Abhishek Goud and Ding, Yanran and Park, Hae-Won},
journal   = {IEEE Robotics and Automation Letters},
volume    = {4},
number    = {4},
pages     = {3355--3362},
year      = {2019},
publisher = {IEEE}
}
```

Contact

We would like to improve our code in as many aspects as possible whether it is building new interfaces or improving the existing code base, suggestions are always welcome. Please contact Dr. Hae-Won Park at the following email address haewonpark@kaist.ac.kr with your valuable feedback.

Known Issues and TroubleShooting

- Input Checking for c,cpp interfaces is not available
- qpSWIFT can run into troubles when the input data is extremely ill-conditioned

Tips

- To access the header files of qpSWIFT library

```
get_target_property(qp_headers qpSWIFT::qpSWIFT-static INTERFACE_INCLUDE_DIRECTORIES)
get_target_property(qp_headers qpSWIFT::qpSWIFT-shared INTERFACE_INCLUDE_DIRECTORIES)
```

- It may not always be necessary to use the maximum number of iterations while running the code. 40 iterations usually suffice for most problems

Appendix

11.1 | Compressed Column Storage format

qpSWIFT operates on sparse matrices which are stored in Compressed Column Storage (CCS) format. In this format, an $m \times n$ sparse matrix A that can contain nnz non-zero entries is stored as an integer array of Ajc of length $n+1$, an integer array Air of length nnz and a real array Apc of length nnz.

- The real array Apr holds all the nonzero entries of A in column major format
- The integer array Air holds the rows indices of the corresponding elements in Apr
- The integer array Ajc is defined as
  - $Ajc[0] = 0$
  - $Ajc[i] = Ajc[i-1] +$ number of non-zeros in i^{th} column of A

11.1.1 Example

For the following sample matrix A,

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 2.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix}$$

we have the following CCS representation

$$\begin{aligned} \text{int Ajc} =& \{0, 3, 6, 8, 10\} \\ \text{int Air} =& \{0, 1, 3, 1, 2, 3, 0, 2, 1, 3\} \\ \text{double Apr} =& \{4.5, 3.1, 3.5, 2.9, 1.7, 0.4, 3.2, 3.0, 2.9, 1.0\} \end{aligned}$$

11.2 | Permutation vector

Directly performing LDL^T factorization on a sparse matrix A typically results in fill-in. A fill-in is a non-zero entry in L but not in A. Higher the fill-in, higher is the memory required to store the matrix factors (L and D) as well as the associated floating point operations. To minimize fill-in, permutation matrices are used and the new system

$$PAP^T$$

is factorized. Obtaining a perfect elimination ordering (permutation matrix with least fill-in) is an NP-hard problem. Hence, heuristics are used to compute permutation matrices. Some of the popular ones are nested dissection and minimum degree ordering methods. qpSWIFT uses the Approximate Minimum Degree (AMD) ordering to compute permutation matrix. The user can opt to use other ordering methods as well.

References

[1] A. G. Pandala, Y. Ding, and H. Park, "qpSWIFT: A real-time sparse quadratic program solver for robotic applications," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3355–3362, Oct 2019.

[2] A. Domahidi, E. Chu, and S. Boyd, "Ecos: An socp solver for embedded systems," in *2013 European Control Conference (ECC)*. IEEE, 2013, pp. 3071–3076.