# 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 Programming Solver. qpSWIFT solves quadratic programs of the following form

$$min_x \frac{1}{2} x^T P x + c^T x$$
$$s.t.\, Ax = b$$
$$Gx \leq h$$

Here, it is assumed that $P$ is a symmetric, positive semi-definite matrix, and $A$ is of 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 the 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 (in progress ...)

# Download

qpSWIFT can be downloaded from the following ⦿ link as

```
git clone https://github.com/qpSWIFT/qpSWIFT
```

# Installation

## 4.1 | C/C++

### 4.1.1 Linux

To build the qpSWIFT library from source on your system, download the qpSWIFT repository and 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
```

You can now add qpSWIFT into your project. Instructions for this can be found in Section - 4.1.4 . Take a look at the demo files in Section - 5.1

### 4.1.2 macOS

To build the qpSWIFT library from source on your system, download the qpSWIFT repository and 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
```

You can now add qpSWIFT into your project. Instructions for this can be found in Section - 4.1.4 . Take a look at the demo files in Section - 5.1

### 4.1.3 Windows

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

**Prerequisites**

- ⊘ cmake
- ⊘ c, c++ compiler

```
    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
```

You can now add qpSWIFT into your project. Instructions for this can be found in Section - 4.1.4 . Take a look at the demo files in Section - 5.1

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

## 4.2 | Matlab

To build the qpSWIFT matlab interface from source on your system, download the qpSWIFT repository and change the matlab working directory to qpSWIFT/matlab. Type the following command in the command window from the qpSWIFT matlab directory

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

**Prerequisites**

⊘ matlab compatible c compiler

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. You can run the `demoqp.m` file to check if the installation was successful. Instructions on using the qpSWIFT matlab interface can be found at 5.2

## 4.3 | Python

To build the qpSWIFT python interface from source on your system, type the following command from the qpSWIFT python directory in the system command line.

```
python setup.py install
```

**Prerequisites**

⊘ c compiler

⊘ numpy

⊘ distutils

Depending on the system, you may require administrator privileges. This builds and installs qpSWIFT module in your system. You can run the `demoqp.py` file to check if the installation is successful. Instructions on using the qpSWIFT python interface can be found at 5.3

## 4.4 | Simulink

There are two Simulink s-functions available for qpSWIFT interface. The first s-function `qpSWIFT_sfunc_e.c` provides an interface to qpSWIFT to handle QP's with both equality and inequality constraints. The second s-function provides the interface to qpSWIFT to handle QP's with only inequality constraints. Before, compiling the s-function, change the number of output variables a.k.a length of the solution of the QP. This can be done by changing the pre-processor definition NV (line 13) to required values in the respective s-functions.

```
#define NV 3 /* change 3 to required value */
```

To compile qpSWIFT simulink interface with both equality and inequality constraints, type

```
Swift_make('qpSWIFT_sfunc_e')
```

To compile qpSWIFT simulink interface with only inequality constraints, type

```
Swift_make('qpSWIFT_sfunc')
```

Demo simulink models can be found at `demoqp_e.slx` and `demoqp.slx`

# Demos

We use the following two quadratic programs as an example on each of the interfaces 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

$$
\min_x \frac{1}{2} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}^T \begin{bmatrix} 5 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}
$$

$$
s.t. \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 3
$$

$$
\begin{bmatrix} -4 & -4 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} -1 \\ -1 \end{bmatrix}
$$

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

### 5.0.2 Problem Setup - 2

The following problem shows a standard QP with only inequality constraints

$$
\min_x \frac{1}{2} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}^T \begin{bmatrix} 5 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}
$$

$$
s.t. \begin{bmatrix} -4 & -4 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} -1 \\ -1 \end{bmatrix}
$$

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

## 5.1 | C/C++ Interface

### 5.1.1 C/C++ Interface via Sparse Matrices

The demo file for C/C++ interface via sparse matrices can be found in `demo/runqp.c`. To interface with qpSWIFT, 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 basic algorithmic skeleton for qpSWIFT is as follows

```
QP *myQP;
myQP = QP_SETUP(n, m, p, Pjc, Pir, Ppr, Ajc, Air, Apr, Gjc, Gir, Gpr, c, h,b,
                sigma_d, Permut);
/* settings can be changed at this point */
qp_int ExitCode = QP_SOLVE(myQP);
/* solution and solution statistics can be accessed now */
QP_CLEANUP(myQP);
```

Initially, a pointer for the QP structure is defined. The data fields of the QP structure and the necessary memory are allocated using the first function `QP_SETUP`. Any custom settings, for example, the maximum number of iterations, can be set after invoking the `QP_SETUP` function. The second function `QP_SOLVE` computes the actual solution. At this stage, the solution is accessible using `myQP->x`. The third function `QP_CLEANUP` clears the entire setup. The description of the input arguments for each of the functions is provided below.

The first function, `QP_SETUP`, has the following input 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 Section - 11. If there are no equality constraints in the QP problem, then set the arguments `Ajc, Air, Apr, B` to a `NULL` pointer and the number of equality constraints as `p = 0`. After invoking this function, the solver settings for the QP can now be changed using

```
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` contains 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 a.k.a solution of the QP */
    myQP->y          /* Dual Solution of the QP (equality constraints) */
    myQP->z          /* Dual Solution of the QP (inequality constraints) */
    myQP->s          /* Slack Variables of the QP */
```

The statistics of the solution can be accessed via `myQP->stats` structure and has the following fields

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

    /* 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 successful   */
                      /* <0 means unsuccessful  */
                      /* -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/C++ Interface via Dense Matrices

The demo file for C++ interface can be found at `demo/runqpcpp.cpp`. The dense interface is also similar to the sparse interface, after adding the header file

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

The algorithmic skeleton for qpswift dense interface is as follows

```
    QP *myQP;
    myQP = QP_SETUP_dense(n, m, p, P, A, G, c, h, b, Permut, COLUMN_MAJOR_ORDERING);
    /* settings can be changed now */
    qp_int ExitCode = QP_SOLVE(myQP);
    /* solution and solution statistics can be accessed now */
    QP_CLEANUP_dense(myQP);
```

The solver can be interfaced via the same three functions, the same way as the sparse interface or can be done via a dense interface using `QP_SETUP_dense` function. Initially, a pointer for the QP structure is defined. The data fields of the QP structure and the necessary memory are allocated using the first function `QP_SETUP_dense`. Any custom settings, for example, the maximum number of iterations, can be set after invoking the `QP_SETUP_dense` function. The second function `QP_SOLVE` computes the actual solution. At this stage, the solution is accessible using `myQP->x`. The third function `QP_CLEANUP_dense` clears the entire setup. The description of the input arguments for each of the functions is provided below.

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 - column 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. If there are no equality constraints, set the A matrix to NULL pointer and the number of equality constraints p to be 0. The 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 Section - 11

After invoking this function, the solver settings for the QP can now be changed using

```
    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 contains 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 a.k.a solution of the QP */
    myQP->y          /* Dual Solution of the QP (equality constraints) */
    myQP->z          /* Dual Solution of the QP (inequality constraints) */
    myQP->s          /* Slack Variables of the QP */
```

The statistics of the solution can be accessed via myQP->stats structure and has the following fields

```
    /* Time Statistics */
    tsetup           /* Setup time ; includes initialisation as well */
    tsolve           /* QP solve time */
```

```
    kkt_time          /* kkt matrix inversion time */
    ldl_numeric       /* kkt matrix factorization time */
    /* Time Statistics */

    /* 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 successful  */
                      /* <0 means unsuccessful  */
                      /* -3 means unused */
```

The last function to call is `QP_CLEANUP_dense`.

```
    QP_CLEANUP_dense(myQP)
```

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

### 5.1.3   Integration with Eigen

qpSWIFT can be interfaced with Eigen via the dense matrix interface. Assuming that the matrices $P, A, G$ and the vectors $c, h, b$ are Eigen matrices, we can invoke qpSWIFT via

```
    myQP = QP_SETUP_dense(P.rows(),A.rows().G.rows(),P.data(),G.data(),A.data(),...
                      c.data(),h.data(),b.data(),NULL,COLUMN_MAJOR_ORDERING);
```

The rest of the steps are similar to the dense interface.

## 5.2 | Matlab Interface

The demo file can be found in `matlab/demoqp.m`. To use qpSWIFT for general quadratic programs of the form

$$min_{x} \frac{1}{2}x^T P x + c^T x$$
$$s.t.\, Ax = b$$
$$Gx \leq h$$

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 programs of the form

$$\min_x \frac{1}{2}x^T P x + c^T x$$
$$s.t.\, Gx \le h$$

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 the 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
-> OUTPUT  : output levels || 1 -- sol + basicInfo
                           || 2 -- sol + basicInfo + advInfo
                           ||   (otherwise) --sol
```

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 factorizing KKT matrix
                 : 2 : maximum number of iterations reached
                 : 3 : unknown problem in solver
```

```
        -> Iterations : number of iterations
        -> Setup Time : setup time (involves setting up QP, solving initial guess)
        -> Solve Time : solution time


   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

The 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 the 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 keys
     -> 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
     -> OUTPUT  : OUTPUT LEVELS ||  1    -- sol + basicInfo
                                ||  2    -- sol + basicInfo + advInfo
                                ||  otherwise   -- sol
```

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 factorizing KKT matrix
                         : 2 : Maximum Number of Iterations Reached
                         : 3 : Unknown Problem in Solver
          -> Iterations : number of iterations
          -> Setup Time : setup time (involves setting up QP and solving initial guess)
          -> Solve Time : solution time

    *    [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

The demo simulink models can be found at `simulink/demoqp_e.slx` and `simulink/demoqp.slx`. Input to the s-function can be changed by modifying the vectors in_P, c, in_G and, h in the matlab function `InputData` of the `demoqp.slx` model . Similar changes can be made to `demoqp_e.slx` file. There are five outputs to each of the simulink s-function which are as follows

```
    solution   /* Solution of QP; dimension is set in s-function (line 13) */
    exitflag   /*  : 0 : optimal solution found
                    : 1 : failure in factorizing KKT matrix
                    : 2 : maximum number of iterations reached
                    : 3 : unknown problem in solver */
    iterations /* number of iterations */
    setuptime  /* setup time */
    solvetime  /* solution time */
```

Please note the following points while using the simulink interface.

▸ Inputs to the simulink s-functions must be vectors of appropriate dimensions. The matrices must be in column-major ordering.

▸ The solution vector of QP is set to 3-dimensions (by default). To get the complete solution of QP, change the pre-processor definition *#define NV* to desired value before compiling the s-function

▸ To use the simulink function in SLRT framework, add all the source files found in `src/` to the `Configuration Parameters > Simulation Target > Custom Code > Source file` block in the simulink model file. Similarly, add the header file directory `include/` to the `Configuration Parameters > Simulation Target > Custom Code > Header file` block

▸ The settings of the QP can be changed in the simulink s-function lines (142-152). Necessary details are found in the comments of the respective s-function files.

# Acknowledgement 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.

# 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 us with your valuable feedback.

**Abhishek Pandala**

Graduate Research Assistant

HDSRL lab

Department of Mechanical Engineering

Virginia Polytechnic Institute and State University

Email ID: agp19@vt.edu

**Yanran Ding**

Postdoctoral Associate

Biomimetic Robotics Lab

Department of Mechanical Engineering

Massachusetts Institute of Technology (MIT)

Email ID: yanran@mit.edu

**Hae-Won Park**

Assistant Professor

Mechanical Engineering

Director, Humanoid Robot Research Center

Korea Advanced Institute of Science and Technology (KAIST)

Email ID: haewonpark@kaist.ac.kr

# Known Issues

🐞 Input Checking for C and C++ interfaces is not done. It is assumed that the inputs provided are in the correct format

🐞 qpSWIFT can run into trouble when the input data is extremely ill-conditioned. Please make sure that the matrices are well-conditioned

🐞 Currently, infeasibility detection is not performed in qpSWIFT. For infeasible problems, qpSWIFT gives an error of maximum iterations reached.

# 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 is usually sufficient for most problems

💡 It is always recommended to compile qpSWIFT library in "release" mode for maximum performance

# 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 contains $nnz$ 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

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

## 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 $A$. The higher the fill-in, the higher 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 the permutation matrix. The user can opt to use other ordering methods as well by passing the argument `Permut` in each of the interfaces.

# 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.