User Manual

Conic Optimization Solver v0.01a

Wolf Optimization

Revision History

Revision	**Date**	**Author(s)**	**Description**
0.01a	2021-Sep-21	NCP	Initial α release of Wolf

Contents

Chapter 1

Introduction

This document provides an overview of the package functionality and a description of the methods implemented.

1.1 Background

The conic optimization solver solves conic programs using a homogeneous primal-dual path-following algorithm. The primal and dual conic programs are of the form

$$\begin{aligned} \text{minimize } & c^T x \\ \text{subject to } & Ax = b \\ & x \in \mathcal{K} \end{aligned} \tag{1.1}$$

and

$$\begin{aligned} \text{maximize } & b^T y \\ \text{subject to } & A^T y + s = c \\ & s \in \mathcal{K}^* \end{aligned} \tag{1.2}$$

where x are the primal variables, $Ax = b$ are the primal constraints, $x \in \mathcal{K}$ are the conic constraints, y are the dual variables, s are the dual slacks, and $s \in \mathcal{K}^*$ represent the dual conic constraints. In order to be explicit in the following, the primal conic program is expanded to show the division between the conic constraints explicitly:

$$\begin{aligned} \text{minimize } & c_L^T x_L + c_Q^T x_Q + c_R^T x_R + c_P^T x_P + c_F^T x_F \\ \text{subject to } & A_L x_L + A_Q x_Q + A_R x_R + A_F x_F = b \\ & x_L \in \mathbb{R}_+^{n_L} \\ & x_Q = (x_Q^{(1)}; ...; x_Q^{(N_Q)}), x_Q^{(k)} \in \mathcal{Q}^{n_Q^k}, k = 1, ..., N_Q, \\ & x_R = (x_R^{(1)}; ...; x_R^{(N_R)}), x_R^{(k)} \in \mathcal{R}^{n_R^k}, k = 1, ..., N_R \\ & x_P = (x_P^{(1)}; ...; x_P^{(N_P)}), x_P^{(k)} \in \mathcal{P}_{\alpha_k}, k = 1, ..., N_P \\ & x_F \in \mathbb{R}^{n_F} \end{aligned} \tag{1.3}$$

where, obviously, $A = [A_L, A_Q, A_Q, A_P, A_F] \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$,$c = (c_L; c_Q; c_R; c_P; c_F) \in \mathbb{R}^n$, $\mathbb{R}_+$ is the set of non-negative real numbers, $\mathcal{Q}^d$ is the Lorentz or quadratic cone of dimension d, $\mathcal{R}^d$ is the rotated quadratic cone of dimension d, $\mathcal{P}_\alpha$ is the three-dimensional power cone with exponent α, n_L is the number of linear variables (or the size of the linear cone), N_Q the number of second-order cones, N_R is the number of rotated quadratic cones, N_P is the number of three-dimensional power cones, and n_F is the number of free variables (or the size of the free cone).

The quadratic cone is defined as the set

$$\mathcal{Q}^d := \{x = (x_1, \bar{x})^T \in \mathbb{R}^d, d \geq 2 : x_1 \geq \sqrt{\bar{x}^T \bar{x}}\}, \tag{1.4}$$

the rotated quadratic cone is defined as the set

$$\mathcal{R}^d := \{x = (x_1, x_2, \tilde{x})^T \in \mathbb{R}^d, , d \geq 3 : 2x_1 x_2 \geq \tilde{x}^T \tilde{x}, x_1 \geq 0, x_2 \geq 0\}, \tag{1.5}$$

and the power cone is defined as the set

$$\mathcal{P}_\alpha := \{x = (x_1, x_2, x_3)^T, 0 < \alpha < 1 : x_1^\alpha x_2^{1-\alpha} \geq |x_3|, x \in \mathbb{R}^3, x_1 \geq 0, x_2 \geq 0\}. \tag{1.6}$$

Note that each variable can only and must be associated with one cone constraint. Additionally, each cone constraint may consist of a number of primitive cone constraints of that type, so that there may be N_Q conic quadratic constraints each of size n_Q^k, giving $\sum_{k=1}^{N_Q} n_Q^k$ total conic quadratic variables.

The expanded dual to (1.3) is

$$
\begin{aligned}
\text{maximise } & b^T y \\
\text{subject to } & A_L^T y + s_L = c_L \\
& A_Q^T y + s_Q = c_Q \\
& A_R^T y + s_R = c_R \\
& A_P^T y + s_P = c_P \\
& A_F^T y = c_F \\
& s_L \in \mathbb{R}_+^{n_L} \\
& s_Q = (s_Q^{(1)}; ...; s_Q^{(N_Q)}), s_Q^{(k)} \in \mathcal{Q}^{n_Q^k}, k = 1, ..., N_Q \\
& s_R = (s_R^{(1)}; ...; s_R^{(N_R)}), s_R^{(k)} \in \mathcal{R}^{n_R^k}, k = 1, ..., N_R \\
& s_P = (s_P^{(1)}; ...; s_P^{(N_P)}), s_P^{(k)} \in \mathcal{P}_{\alpha_k}^*, k = 1, ..., N_P
\end{aligned}
\tag{1.7}
$$

where $y \in \mathbb{R}^m$, and $\mathcal{P}_\alpha^*$ is the dual cone to $\mathcal{P}_\alpha$ defined as

$$
\mathcal{P}_\alpha^* := \{s = (x_1, x_2, x_3)^T, 0 < \alpha < 1 : s^T x \geq 0 \, \forall \, x \in \mathcal{P}_\alpha\}. \tag{1.8}
$$

The remaining cones are *symmetric*, which has the property that their dual cone is the same as the original cone. Much of the initial research upon which the solver is based is covered in Reference [6].

1.2 The conic optimization solver algorithm

The implemented algorithm is a simplified homogeneous self-dual predictor-corrector primal-dual path-following interior-point method that traces a path through the neighbourhood of the central path towards an optimal point. All iterates satisfy the cone constraints but the feasibility in terms of the equality constraints is generally improved at each iteration at approximately the same rate as the reduction in the average complementarity gap, μ.

The algorithm proceeds a follows:

```
1. Set the initial point.
2. for $1 \leq i \leq i_{max}$
3.  Build the Schur complement.
4.  Compute the affine-scaling search direction.
5.  Determine $\gamma$ for combined centering-corrector direction.
6.  Compute the combined centering-corrector search direction.        (1.9)
7.  Determine the step length, $\alpha$.
8.  Update the variables.
9.  Check for convergence and exit if achieved.
10. end
```

1.2.1 Presolving the problem

An effective presolve routine can provide significant reductions in runtime, as well as improve the characteristics of the problem being solved without altering the essence of the problem. The various components of a presolve routine are covered below.

Eliminating free variables

Free variables often present numerical difficulties in the interior point method and complicate the computation of the search direction (requiring a symmetric indefinite solver or regularisation which affects the conditioning of the coefficient matrix). Thus, it is worthwhile eliminating any free variables from the problem where it can be done so without severely affecting the sparsity of the constraint matrix.

Denote the columns of A associated with the free variables as A_F and the remainder as $\bar{A}$, so $A = \begin{bmatrix} \bar{A} & A_F \end{bmatrix}$. If we can the find a non-singular basis of A_F, we can eliminate those columns and an equal number of rows from A. Call the basis we identify in A_F A_B and let the remaining columns of A_F be $\tilde{A_F}$ so $A_F = \begin{bmatrix} \tilde{A_F} A_B \end{bmatrix}$. Let $A_N = \begin{bmatrix} \bar{A} & \tilde{A_F} \end{bmatrix}$. We now have

1.2.2 The initial point

The initial point is chosen to satisfy the cone constraints as the minimum of $\frac{1}{2}x^T x + F(x)$, that is, where $x = -F'(x)$ and $F(x)$ is the primal barrier function. Specifically, each of the linear variables, $x_L^{(k)}$ and $s_L^{(k)}$, are set to 1. For each primitive conic quadratic constraint, $x_1^{(k)} = s_1^{(k)} = \sqrt{2}$ and $\bar{x}^{(k)} = \bar{s}^{(k)} = \{0\}$. The rotated quadratic conic variables are $x_1^{(k)} = x_2^{(k)} = s_1^{(k)} = s_2^{(k)} = 1$ and $\tilde{x}^{(k)} = \tilde{s}^{(k)} = \{0\}$. The initial power cone variable values are $x_1^{(k)} = \sqrt{1+\alpha}, x_2^{(k)} = \sqrt{2-\alpha}, x_3^{(k)} = 0$ and $s^{(k)} = x^{(k)}$. This also has the effect that $\mu_0 = \frac{x^T s}{\nu} = 1$, where ν is the barrier parameter (1 for linear cone, 2 for the quadratic cones, and 3 for the power cone).

1.2.3 Scaling

The Nesterov-Todd scaling [5, 1] is used for the symmetric cones, while the Tunçel scaling [8] is used for the non-symmetric cones.

For each symmetric cone, the scaling of x is made to equal the inverse scaling of s. The non-symmetric cone scaling formulations, however, lead to slightly different inverse scalings.

For linear variables, the scaling is defined as $d_j = \sqrt{s_j/x_j}$ so that $d_j x_j = s_j/d_j = \sqrt{x_j s_j} = v_j^L$.

For the second-order cone variables, we set the scaling matrix W such that $W_j x_j = W_j^{-1} s_j = v_j^Q$. This scaling matrix can be constructed by letting

$$w_j = \begin{Bmatrix} w_1 \\ w_{2:n_j} \end{Bmatrix} = \frac{1}{\sqrt{2\left(\gamma(x_j)\gamma(s_j) + x^T s\right)}} \begin{Bmatrix} \theta_j^{-1} s_1 + \theta_j x_1 \\ \theta_j^{-1} s_{2:n_j} - \theta_j x_{2:n_j^Q} \end{Bmatrix},$$

where $\gamma(x_j^Q) = \sqrt{x_1^2 - \|x_{2:n_j}\|^2}$ and $\theta_j = \sqrt{\gamma(s_j)/\gamma(x_j)}$.

1.2.4 The search direction

An affine-scaling (predictor) direction is computed first and then used to determine the right hand side for the combined centering-corrector direction. The process to compute the predictor and combined centering-corrector directions are very similar, both being based on solving the following system of equations.

$$\begin{bmatrix} A & -b & & & \\ & -c & A^T & I & \\ -c^T & & b^T & & -1 \\ E & & & F & \\ & \kappa & & & \tau \end{bmatrix} \begin{Bmatrix} d_x \\ d_\tau \\ d_y \\ d_s \\ d_\kappa \end{Bmatrix} = \begin{Bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{Bmatrix}. \tag{1.10}$$

Here, the right hand side for the predictor is

$$\begin{Bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4^P \\ r_5^P \end{Bmatrix} = \begin{Bmatrix} (\gamma - 1)(Ax - b\tau) \\ (\gamma - 1)\left(A^T y + s - c\tau\right) \\ (\gamma - 1)\left(b^T y - c^T x - \kappa\right) \\ \gamma\mu e - \operatorname{svec}\left(H_G(XS)\right) \\ \gamma\mu - \tau\kappa \end{Bmatrix}, \tag{1.11}$$

(with $\gamma = 0$) which differs to the combined centering-corrector direction only in the term for r_4

$$\begin{Bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4^C \\ r_5^C \end{Bmatrix} = \begin{Bmatrix} (\gamma - 1)(Ax - b\tau) \\ (\gamma - 1)\left(A^T y + s - c\tau\right) \\ (\gamma - 1)\left(b^T y - c^T x - \kappa\right) \\ \gamma\mu e - \operatorname{svec}\left(H_G(XS) + H_G(D_x D_s)\right) \\ \gamma\mu - \tau\kappa - d_\tau d_\kappa \end{Bmatrix}. \tag{1.12}$$

1.2.5 Computing the search direction

The approach usually taken is to reduce this system to either the augmented equations or the Schur complement equation. This is done as follows. First, $d_\kappa = \frac{1}{\tau}(r_5 - \kappa d_\tau)$ and $d_s = F^{-1}(r_4 - E d_x)$ are eliminated, leaving

$$\begin{bmatrix} -F^{-1}E & A^T & -c \\ A & 0 & -b \\ -c^T & b & \kappa/\tau \end{bmatrix} \begin{Bmatrix} d_x \\ d_y \\ d_\tau \end{Bmatrix} = \begin{Bmatrix} r_2 - F^{-1} r_4 \\ r_1 \\ r_3 + r_5/\tau \end{Bmatrix}.$$

We can then eliminate d_x and d_y from the last row. Taking

$$\begin{Bmatrix} d_x \\ d_y \end{Bmatrix} = \begin{bmatrix} -F^{-1}E & A^T \\ A & 0 \end{bmatrix}^{-1} \left(\begin{Bmatrix} r_2 - F^{-1} r_4 \\ r_1 \end{Bmatrix} + d_\tau \begin{Bmatrix} c \\ b \end{Bmatrix} \right), \tag{1.13}$$

we get

$$\left(\frac{\kappa}{\tau} + \begin{Bmatrix} -c \\ b \end{Bmatrix}^T \begin{bmatrix} -F^{-1}E & A^T \\ A & 0 \end{bmatrix}^{-1} \begin{Bmatrix} c \\ b \end{Bmatrix} \right) d_\tau = r_3 + \frac{r_5}{\tau} - \begin{Bmatrix} -c \\ b \end{Bmatrix}^T \begin{bmatrix} -F^{-1}E & A^T \\ A & 0 \end{bmatrix}^{-1} \begin{Bmatrix} r_2 - F^{-1} r_4 \\ r_1 \end{Bmatrix}. \tag{1.14}$$

If we obtain g_1, g_2, h_1 and h_2 from

$$\begin{bmatrix} -F^{-1}E & A^T \\ A & 0 \end{bmatrix} \begin{Bmatrix} g_1 \\ g_2 \end{Bmatrix} = \begin{Bmatrix} c \\ b \end{Bmatrix} \tag{1.15}$$

and

$$\begin{bmatrix} -F^{-1}E & A^T \\ A & 0 \end{bmatrix} \begin{Bmatrix} h_1 \\ h_2 \end{Bmatrix} = \begin{Bmatrix} r_2 - F^{-1}r_4 \\ r_1 \end{Bmatrix}, \tag{1.16}$$

we can compute

$$d_\tau = \frac{r_3 + r_5/\tau + c^T h_1 - b^T h_2}{\kappa/\tau - c^T g_1 + b^T g_2}. \tag{1.17}$$

We have thus reduced the problem to solving the augmented equations (1.15) and (1.16). This leads to the two equations

$$AE^{-1}FA^T g_2 = b + AE^{-1}Fc \tag{1.18}$$

and

$$AE^{-1}FA^T h_2 = r_1 + A\left(E^{-1}Fr_2 - E^{-1}r_4\right). \tag{1.19}$$

For SDP, the right hand side for the predictor direction is equivalent to

$$A(G \otimes_s G)\text{svec}(G^T \text{mat}(r_2)G + D) \tag{1.20}$$

and

$$A(G \otimes_s G)\text{svec}(G^T \text{mat}(r_2)G - \gamma\mu D^{-1} + D) \tag{1.21}$$

With g_2 and h_2, we can compute g_1 and h_1 fom the first block equation of (1.15) and (1.16):

$$\begin{aligned} g_1 &= E^{-1}F\left(A^T g_2 - c\right) && (1.22) \\ h_1 &= E^{-1}F\left(A^T h_2 - r_2 + F^{-1}r_4\right) && (1.23) \end{aligned}$$

. From (1.13) we recover d_x and d_y,

$$\begin{Bmatrix} d_x \\ d_y \end{Bmatrix} = \begin{Bmatrix} h_1 \\ h_2 \end{Bmatrix} + d_\tau \begin{Bmatrix} g_1 \\ g_2 \end{Bmatrix}. \tag{1.24}$$

To avoid increasing dual infeasibility, we use the second block equation from (1.10) to get d_s,

$$d_s = r_2 + cd_\tau - A^T d_y. \tag{1.25}$$

Finally, d_κ is computed from the last row of (1.10)

$$d_\kappa = \frac{r_5 - \kappa d_\tau}{\tau}. \tag{1.26}$$

Assembling the Schur complement

Before solving (1.18) and (1.19), we must assemble and factorize the coefficient matrix known as the Schur complement, $AE^{-1}FA^T$. The approach for both the semidefinite and the second-order cones are different to that of the linear and free cones.

Columns of A associated with a linear or free variable For the linear and regularized free variables, $E^{-1}F$ is diagonal and easily computed. This makes it possible to perform $n_L + n_F$ symmetric rank 1 outer product updates to the Schur complement:

$$M_L + M_F = \left(\sum_{j=1}^{n_L} \theta_{Lj}^{-2} a_{Lj} a_{Lj}^T \right) + \epsilon^{-1} \left(\sum_{j=1}^{n_F} a_{Fj} a_{Fj}^T \right),$$

where ϵ is the regularization parameter.

Columns of A associated with a variable constrained by a second-order cone $E^{-1}F$ for the kth second-order cone can be written as $Q^k - 2Q^k w^k w^k \left(Q^k\right)^T$, where Q is a diagonal matrix with entries ± 1. This means that we perform a symmetric rank 1 outer product for each cone and another for each column associated with second-order cone, giving $n_Q^k + 1$ symmetric rank 1 outer products:

$$M_Q^k = \theta_{Qk}^{-2} \left(a_{Qj}^k \left(a_{Qj}^k\right)^T - \sum_{j=2}^{n_Q^k} a_{Qj}^k \left(a_{Qj}^k\right)^T \right) + 2\theta_{Qk}^{-2} \left(A_Q^k Q^k w^k\right) \left(A_Q^k Q^k w^k\right)^T.$$

Each of the M_Q^k are summed to form M_Q.

Exploiting fixed variables subject to a second-order cone constraint

If a free variable or linear variable is fixed when considering one or more of the constraint equations, then it may be substituted out of the problem (assuming, if it is a linear variable, that it is non-negative). If the variable is associated with a second-order cone constraint, however, then it is not as straightforward. Starting from (1.15) and letting $H = F^{-1}E$, we partition the block 2×2 system (and scale if necessary) so that

$$\begin{bmatrix} -H_{11} & -H_{12} & 0 & I \\ -H_{21} & -H_{22} & A_{12}^T & 0 \\ 0 & A_{12} & 0 & 0 \\ I & 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} g_1^{(1)} \\ g_1^{(2)} \\ g_2^{(1)} \\ g_2^{(2)} \end{Bmatrix} = \begin{Bmatrix} c^{(1)} \\ c^{(2)} \\ b^{(1)} \\ b^{(2)} \end{Bmatrix} \tag{1.27}$$

and the entries of g_1 that are fixed in the first partition, $g_1^{(1)}$. From the fourth block row, we have $g_1^{(1)} = b^{(2)}$. We then consider the second and third block equations

$$\begin{bmatrix} -H_{22} & A_{12}^T \\ A_{12} & 0 \end{bmatrix} \begin{Bmatrix} g_1^{(2)} \\ g_2^{(1)} \end{Bmatrix} = \begin{Bmatrix} c^{(2)} + H_{21} b^{(2)} \\ b^{(1)} \end{Bmatrix}. \tag{1.28}$$

We can then reduce this system to the Schur complement by noting that

$$H_{22} = Q_{22} + 2 w_2^T w_2^T \tag{1.29}$$

and, considering that for a second-order cone Q is a diagonal matrix and $Q^{-1} = Q$ which holds for any submatrix also, we can use the Sherman-Morrison-Woodbury formula to get

$$H_{22}^{-1} = -Q_{22} - 2 \frac{Q_{22} w_2 w_2^T Q_{22}}{1 - 2 w_2^T Q_{22} w_2}. \tag{1.30}$$

This form is very similar to that of $H^{-1} = E^{-1} F$ and so the Schur complement can be constructed just as cheaply. So we then solve (1.28) by computing the Schur complement and solving

$$A_{12} H_{22}^{-1} A_{12}^T g_2^{(1)} = b^{(1)} + A_{12} H_{22}^{-1} \left(c^{(2)} + H_{21} b^{(2)} \right) \tag{1.31}$$

for $g_2^{(1)}$, and then using the same explicit form for H_{22}^{-1} to obtain $g_1^{(2)}$ from

$$H_{22} g_1^{(2)} = A_{12}^T g_2^{(1)} - c^{(2)} - H_{21} b^{(2)}. \tag{1.32}$$

Finally, we compute $g_2^{(2)} = c^{(1)} + H_{11} b^{(2)} + H_{12} g_1^{(2)}$.

The process is identical to solve (1.16) albeit with a different right-hand side. Both systems require three operations with the partitioned $F^{-1}E$; first the matrix-vector multiply of the form $H_{21} b^{(2)}$, then another with the inverse of H_{22} in the form $H_{22}^{-1} c^{(2)}$, and finally $H_{11} g_1^{(1)} + H_{12} g_1^{(2)}$.

Handling dense columns

In some problems, dense columns are present in the constraint matrix, A. This can lead to a significantly more dense Schur complement matrix than would otherwise be the case. It is thus common practice to avoid including the dense column when forming the Schur complement matrix [2, 4]. The approach used here varies in the implementation described in the literature so it can be used for variables in the conic quadratic, linear, and free cones.

The approach starts with by considering augmented equations with the dense columns of A arranged so as A_D holds the dense columns and A_0 is the remained of A. These dense columns are identified by comparing the number of entries in each column against the average column count. If a column is above a user-specified multiple of the average, then the column is deemed dense. The $(1,1)$ block is arranged to match the split. We then have

$$\begin{bmatrix} -H_0 & & A_0^T \\ & -H_D & A_D^T \\ A_0 & A_D & \end{bmatrix} \begin{Bmatrix} g_1^{(0)} \\ g_1^{(D)} \\ g_2 \end{Bmatrix} = \begin{Bmatrix} c^{(0)} \\ c^{(D)} \\ b \end{Bmatrix}. \tag{1.33}$$

By eliminating $g_1^{(0)}$, we can proceed to solve the 2×2 system that is now of order $m + n_D$, where n_D is the number of dense columns:

$$\begin{bmatrix} -H_D & A_D^T \\ A_D & M \end{bmatrix} \begin{Bmatrix} g_1^{(D)} \\ g_2 \end{Bmatrix} = \begin{Bmatrix} c^{(D)} \\ b + A_0 H_0^{-1} c^{(0)} \end{Bmatrix}, \tag{1.34}$$

where $M = A_0 H_0^{-1} A_0^T$. Eliminating $g_2 = M^{-1}\left(b + A_0 H_0^{-1} c^{(0)} - A_D g_1^{(D)}\right)$ now gives us

$$-\left(H_D + A_D^T M^{-1} A_D\right) g_1^{(D)} = c^{(D)} - A_D^T M^{-1}\left(b + A_0 H_0^{-1} c^{(0)}\right). \tag{1.35}$$

Although there are three places where M^{-1} is used, if we consider the same equation but with the Cholesky factorization $M = LL^T$, we have

$$-\left(H_D + A_D^T L^{-T} L^{-1} A_D\right) g_1^{(D)} = c^{(D)} - A_D^T L^{-T} L^{-1}\left(b + A_0 H_0^{-1} c^{(0)}\right), \tag{1.36}$$

and we see that we only need to do three triangular solves (that is, with L) rather than the three full solves with M, with $g_2 = L^{-T} L^{-1}\left(b + A_0 H_0^{-1} c^{(0)} - A_D g_1^{(D)}\right) = L^{-T}\left(r - V g_1^{(D)}\right)$.

We should take into account, however, that we are going to solve for two different right hand sides in computing the predictor direction for the homogeneous self-dual embedding and one for the combined centering-corrector. For simplicity, we describe solving with just one right hand side. The steps are as follows:

1. Compute the Cholesky factorization $LL^T = M$.

2. Solve $LV = A_D$ for V.

3. Solve $Lr = \left(b + A_0 H_0^{-1} c^{(0)}\right)$ for r.

4. Solve $\left(H_D + V^T V\right) g_1^{(D)} = \left(V^T r - c^{(D)}\right)$ for $g_1^{(D)}$.

5. Solve $L^T g_2 = \left(r - V g_1^{(D)}\right)$ for g_2.

6. Solve $-H_0 g_1^{(0)} = c^{(0)} + A_0^T g_2$ for $g_1^{(0)}$.

After unpermuting we can then continue to compute the search direction for the remaining variables.

Dealing with fixed variables and dense columns simultaneously

If we have both fixed variables and dense columns in the same problem then, merging the above notation, we have

$$\begin{bmatrix} -H_{11} & -H_{12} & -H_{13} & 0 & I \\ -H_{21} & -H_0 & 0 & A_S^T & 0 \\ -H_{31} & 0 & -H_D & A_D^T & 0 \\ 0 & A_0 & A_D & 0 & 0 \\ I & 0 & 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} g_1^{(1)} \\ g_1^{(2)} \\ g_1^{(D)} \\ g_2^{(1)} \\ g_2^{(2)} \end{Bmatrix} = \begin{Bmatrix} c^{(1)} \\ c^{(2)} \\ c^{(D)} \\ b^{(1)} \\ b^{(2)} \end{Bmatrix}. \tag{1.37}$$

First we can eliminate the fixed variables $g_1^{(1)} = b^{(2)}$, then consider the second, third and fourth block rows.

$$\begin{bmatrix} -H_0 & 0 & A_S^T \\ 0 & -H_D & A_D^T \\ A_0 & A_D & 0 \end{bmatrix} \begin{Bmatrix} g_1^{(2)} \\ g_1^{(D)} \\ g_2^{(1)} \end{Bmatrix} = \begin{Bmatrix} c^{(2)} + H_{21} b^{(2)} \\ c^{(D)} + H_{31} b^{(2)} \\ b^{(1)} \end{Bmatrix}. \tag{1.38}$$

This system only differs from the dense column situation above only in the right hand side and so we can follow the same process. From the first block row we then compute $g_2^{(2)} = c^{(1)} + H_{11} g_1^{(1)} + H_{12} g_1^{(2)} + H_{13} g_1^{(D)}$.

1.2.6 Determining the step length

Interior-point methods rely on all iterates maintaining feasibility with respect to their conic constraints. In order to achieve this, checks must be made to ensure that this remains so by limiting the step length in the search direction. The approach used here requires that the iterates remain within some neighbourhood of the central path and follows the basic approach of Andersen et al. [1], which is based on the work of Nesterov and Todd [5] for all cones and has been generalized to non-symmetric cones by Tunçel [8].

Chapter 2

APIs

To use the Wolf conic optimization solver, wrappers in the most common languages have been developed. These wrappers aim to simplify the use and provide a more native feel for each language.

2.1 C/C++

There are two interfaces provided for C/C++. There is a simple interface for solving one off problems and expert interfaces that allow a series of problems with different values to be solved without repeating the initialisation and allocation procedures each time. There are only two files required to use the library from C/C++, the header file describing the interface, `wolf.h`, and the dynamically linked library `wolfcos.dll` and `wolfcos.lib` on Windows or `wolfcos.a` on Linux, BSDs, or OS X.

2.1.1 Simple interface

The simple interface is a collection of three calls.

1. `wolf_setpar(wolf_problem_t *problem);` This sets the default parameters so you can see what parameter values will be used and change them if you desire.

2. `wolf_cos(wolf_problem_t *problem);` This solves the problem.

3. `const char * wolf_geterrormsg(integer error);` This function returns a human readable string associated with the error code.

The type `wolf_problem_t` collects all the information on the problem formulation and the solution produced. The following must be allocated and input values present before calling `wolf_cos()`:

- `wolf_problem_t.m` - the number of rows in the constraint matrix A
- `wolf_problem_t.n` - the number of columns in the constraint matrix A
- `wolf_problem_t.nF` - the number of free variables in the problem
- `wolf_problem_t.nL` - the number of linear variables in the problem
- `wolf_problem_t.nQ` - the number of second-order cones in the problem
- `wolf_problem_t.nR` - the number of rotated second-order cones in the problem
- `wolf_problem_t.nP` - the number of power cones in the problem
- `wolf_problem_t.pQ` - a pointer to the first entry in each second-order cone, with the last entry equal to the position after the last second-order cone variable (i.e. `wolf_problem_t.pQ[wolf_problem_t.nR]-wolf_problem_t.pQ[0]` is the total count of all variables constrained by a second-order cone)
- `wolf_problem_t.pR` - a pointer to the first entry in each rotated second-order cone, with the last entry equal to the position after the last rotated second-order cone variable (i.e. `wolf_problem_t.pR[wolf_problem_t.nR]-wolf_problem_t.pR[0]` is the total count of all variables constrained by a rotated second-order cone)
- `wolf_problem_t.PCalpha` - array of exponents α for each power cone in the problem
- `wolf_problem_t.pA` - array of pointers to the first entry in each column of `iA` and `xA` for the CSC structure of the constraint matrix A
- `wolf_problem_t.iA` - array of row indices for the CSC structure of the constraint matrix A
- `wolf_problem_t.xA` - array of values for the CSC structure of the constraint matrix A
- `wolf_problem_t.b` - the right-hand side vector for the equality constraints b
- `wolf_problem_t.c` - the linear objective function c
- `wolf_problem_t.x` - the primal solution vector x
- `wolf_problem_t.y` - the dual solution vector y

- `wolf_problem_t.s` - the dual slacks vector s

C example (simple interface)

The following code, when compiled, will solve the problem $\min \left\{ x_1 : x_1 \geq \sqrt{x_2^2 + x_3^2}, x_3 = 1 \right\}$.

```
#include "wolf.h"
#include <stdio.h>

static int print_progress(integer iteration, double *dnfo)
{
  /* Output progress update. */
  printf("%3i %8.1e %8.1e %8.1e %8.1e %11.4e %11.4e %7.3f %8.1e
      %8.1e %7.1f\n", (int)iteration, dnfo[0], dnfo[1], dnfo
     [2], dnfo[3], dnfo[4], dnfo[5], dnfo[6], dnfo[7], dnfo[8],
      dnfo[9]);
  return 0;
}
int main()
{
  /* Set up problem struct. */
  wolf_cos_t prob;
  prob.m = 1;
  prob.n = 3;
  prob.nF = 0;
  prob.nL = 0;
  prob.nQ = 1;
  prob.nR = 0;
  prob.nS = 0;
  prob.nP = 0;
  prob.nE = 0;

  integer pQ[2] = { 0, 3 };
  integer pA[4] = { 0, 0, 0, 1 };
  integer iA[1] = { 0 };
  double xA[1] = { 1.0 };
  double b[1] = { 1.0 };
  double c[3] = { 1.0, 0.0, 0.0 };

  double x[3], y[1], s[3];

  prob.x = x;
  prob.y = y;
  prob.s = s;
prob.pQ = pQ;
prob.pA = pA;
prob.iA = iA;
prob.xA = xA;
prob.b = b;
prob.c = c;

/* Set default parameters. */
wolf_setpar(&prob);

/* Modify default parameters. */
prob.ipar[WOLF_COS_IS_MAXITER] = 99;

/* Output progress header. */
printf("\nWolf Conic Optimization Solver\n\n");
printf(" It      pinf       dinf       ginf        u          pobj
          dobj         alpha      tau        kappa       time\n");

/* Solve the problem. */
wolf_cos(&prob, print_progress);

/* Check for error and display message. */
if (prob.error != 0)
  printf("*** %s ***\n", wolf_geterrormsg(prob.error));

printf("\nSolution info:\n--------------\n");
printf("Average objective: %9.7f\n", 0.5*(prob.dnfo[0] + prob
   .dnfo[1]));
printf("Iterations: %d\n", prob.info[0]);
printf("Primal infeasibility:  %8.1e\n", prob.dnfo[2]);
printf("Dual infeasiblity:     %8.1e\n", prob.dnfo[3]);
printf("Relative duality gap:  %8.1e\n", prob.dnfo[4]);

printf("x = [ %9.7f %9.7f %9.7f ]\n", x[0], x[1], x[2]);
printf("y = [ %9.7f ]\n", y[0]);
printf("s = [ %9.7f %9.7f %9.7f ]\n", s[0], s[1], s[2]);

printf("\nProblem size:\n-------------\n");
printf("Number of constraints: %10li (originally %10li)\n",
   prob.info[1], prob.m);
printf("Number of variables:   %10li (originally %10li)\n",
   prob.info[2], prob.n);
  printf("Second-order cones:     %10li\n", prob.info[5]);
  printf("Non-zeros in A:         %10li (originally %10li)\n",
     prob.info[7], prob.pA[prob.n]);
  printf("Non-zeros in M:         %10li\n", prob.info[8]);
  printf("Non-zeros in L:         %10li\n", prob.info[9]);
}
```

2.2 MATLAB

2.2.1 Installation

To call the Wolf Optimization solver from MATLAB, navigate to the folder containing the files `wolfcos.m`, `mex_wolfcos.c`, `wolf.h` and the library(`wolfcos.dll` and `wolfcos.lib` on Windows, or `wolfcos.a` on Linux and OS X). Run the command

```
>> mex -largeArrayDims mex_wolfcos.c -lwolfcos
```

to compile the driver. Note that the To add the current directory to the path (allowing you to call `wolfcos` from any location), use

```
>> addpath(pwd)
```

and

```
>> savepath
```

to save it so that it is included on the path every time you open MATLAB.

2.2.2 Usage

Quick start

To get going immediately, the following are some examples of small but interesting problems and how to solve them with **wolfcos**. The first is an SOCP from Sturm [7]:

```
>> A=[0 0 1]; b=1; c=[1 -1 0]'; K.q=3; wolfcos(A,b,c,K);
```

A small difficult SOCP presented by Ben-Tal and Nemirovski [3] is

$$\inf \left\{ x_1 | \sqrt{(x_1 - x_2)^2 + 1} \leq x_1 + x_2 \right\}.$$

Let $z_1 = x_1 + x_2$, $z_2 = x_1 - x_2$ and $z_3 = 1$. The objective function is then $\frac{1}{2}(z_1 + z_2) = x_1$, and the problem becomes $\inf \left\{ \frac{1}{2}(z_1 + z_2\} | z_1 \geq \sqrt{z_2^2 + z_3^2}, z_3 = 1 \right\}$. Solve this with

```
>> A=[0 0 1]; b=1; c=[0.5 0.5 0]'; K.q=3; wolfcos(A,b,c,K);
```

A simple but dual infeasible SOCP (again from Ben-Tal and Nemirovski [3]) is

```
>> A=[1 0 -1]; b=0; c=[0 1 0]'; K.q=3; wolfcos(A,b,c,K);
```

Detailed usage instructions

The MATLAB interface uses a SeDuMi-based input format, with constraint matrix `A`, right hand side vector `b`, objective function vector `c`, and the cone structure `K`. The fields of `K` are

- `K.f` is the number of free variables
- `K.l` is the number of linear variables
- `K.q` is a list of the size of each conic quadratic cone, so there are `sum(K.q)` variables and `length(K.q)` second-order cones
- `K.r` is a list of the size of each rotated conic quadratic cone, so there are `sum(K.r)` variables and `length(K.r)` rotated second-order cones
- `K.p` is a two column array, where the first column is the list of the size of each power cone, so there are `sum(K.p(:,1))` variables and `length(K.p(:,1))` power cones, and the second column is the exponent α for each cone

It is also possible to pass the majority of the parameters controlling the presolve through an options structure with

```
>> wolfcos(A,b,c,K,opt);
```

The fields of `opt` that may be specified are as follows:

- `opt.lu_pivotcoleps` - the pivot column threshold for the *LU* factorisation used to eliminate free variables
- `opt.lu_pivotroweps` - the pivot row threshold for the *LU* factoration used to eliminate free variables
- `opt.lu_absdropeps` - the absolute drop tolerance used in the *LU* factorisation to remove very tiny entries from the constraint matrix

- `opt.lu_searchcols` - the number of columns to search for a pivot in the *LU* factorisation used to eliminate free variables
- `opt.lu_maxfillperpivot` - the maximum allowed fill-in per pivot less the number of entries in the pivot row and column
- `opt.presolve_equiliter` - the maximum number of iterations to use to equilibrate the constraint matrix before solving with the IPM, or use -1 for ∞-norm row scaling or -2 for 2-norm row scaling
- `opt.presolve_densecolratio` - any column greater than this parameter times the average number of column entries is deemed to be dense and handled differently in computing the search direction
- `opt.ipm_maxiter` - the maximum number of IPM iterations
- `opt.ipm_maxstagiter` - the maximum number of IPM iterations with a stagnating primal or dual infeasibility
- `opt.ipm_epsP` - the primal infeasibility convergence tolerance
- `opt.ipm_epsD` - the dual infeasibility convergence tolerance
- `opt.ipm_epsG` - the relative duality gap convergence tolerance
- `opt.ipm_epsA` - the significant figure convergence tolerance
- `opt.ipm_epsI` - tolerance used to identify infeasible and ill-posed problems
- `opt.ipm_epsMU` - tolerance used to identify ill-posed problems
- `opt.ipm_hoodbeta` - ∞-norm neighbourhood parameter used in determining the maximum step length
- `opt.ipm_predcorrdelta` - the parameter used to ensure a minimum amount of progress towards an optimal solution is attempted
- `opt.ipm_steprelax` - step length relaxation
- `opt.ipm_freevarreg` - used to regularise the augmented equations for free variables
- `opt.ipm_maxinfeasincratio` - the maximum primal or dual infeasibility ratio increase in one iteration (unless the infeasibility is still less than the threshold)
- `opt.ipm_infeasstagratio` - the ratio of successive infeasibilities deemed to be stagnating

If desired, x, y and s can be returned by **wolfcos**, along with an `info` structure that contains information on the computed solution and the solution process with

```
>> [x,y,s,info] = wolfcos(A,b,c,K);
```

The fields of info are as follows:

1. `info(1)` - the status returned by the solver
2. `info(2)` - the number of IPM iterations
3. `info(3)` - the primal objective value
4. `info(4)` - the dual objective value
5. `info(5)` - the relative primal infeasibility, $\|Ax - b\|_\infty / (1 + \|b\|_\infty)$
6. `info(6)` - the relative dual infeasibility, $\|A^T y + s - c\|_\infty / (1 + \|c\|_\infty)$
7. `info(7)` - the relative duality gap, $|c^T x - b^T y| / \left(1 + \frac{1}{2}\left(|c^T x| + |b^T y|\right)\right)$
8. `info(8)` - time spent in presolve
9. `info(9)` - time spent in initialisation (including ordering routine)
10. `info(10)` - time spent in IPM
11. `info(11)` - time spent in postsolve
12. `info(12)` - total time taken to solve the problem
13. `info(13)` - the total number of equality constraints in the solved problem
14. `info(14)` - the total number of variables in the solved problem
15. `info(15)` - the number of free variables in the solved problem
16. `info(16)` - the number of linear variables in the solved problem
17. `info(17)` - the number of second-order cones in the solved problem
18. `info(18)` - the number of rotated second-order cones in the solved problem
19. `info(20)` - the number of power cones in the solved problem
20. `info(22)` - the number of non-zeros in the constraint matrix
21. `info(23)` - the number of non-zeros in the search direction Schur complement
22. `info(24)` - the number of non-zeros in the Cholesky factorization
23. `info(25)` - number of columns considered dense in the constraint matrix

24. `info(26)` - Schur complement build time
25. `info(27)` - factorization time
26. `info(28)` - search direction solve time
27. `info(29)` - search time
28. `info(30)` - dense solve time

2.2.3 Help

You can also use MATLAB's inbuilt help command with **wolfcos** to get the usage information with

```
>> help wolfcos
```

Bibliography

[1] E. D. Andersen, C. Roos, and T. Terlaky. On implementing a primal–dual interior–point method for conic quadratic optimization. *Mathematical Programming*, 95(2):249–277, 2003.

[2] Knud D. Andersen. A modified Schur–complement method for handling dense columns in interior–point methods for linear programming. *ACM Transactions on Mathematical Software*, 22(3):348–356, sep 1996.

[3] Aharon Ben-Tal and Arkadi Nemirovski. *Lectures on modern convex optimization.* Society for Industrial and Applied Mathematics, jan 2001.

[4] Zhi Cai and Kim-Chuan Toh. Solving second order cone programming via a reduced augmented system approach. *SIAM Journal on Optimization*, 17(3):711–737, jan 2006.

[5] Yu. E. Nesterov and M. J. Todd. Primal–dual interior–point methods for self–scaled cones. *SIAM Journal on Optimization*, 8(2):324–364, may 1998.

[6] Nathan C. Podlich. *The development of efficient algorithms for large–scale finite element limit analysis.* PhD thesis, University of Newcastle, 2017.

[7] Jos F. Sturm. Implementation of interior point methods for mixed semidefinite and second order cone optimization problems. *Optimization Methods and Software*, 17(6):1105–1154, jan 2002.

[8] Levent Tunçel. Generalization of primal–dual interior–point methods to convex optimization problems in conic form. *Foundations of Computational Mathematics*, 1(3):229–254, jul 2001.