HDSDP 1.0

Software for Semidefinite Programming

User Manual

Wenzhi Gao Dongdong Ge

Shanghai University of Finance and Economics

Yinyu Ye

Stanford University

1 Introduction

HDSDP is a numerical software designed for general-purpose semi-definite programming problems (SDP).

$$\begin{aligned} \min_{\mathbf{X}} \quad & \langle \mathbf{C}, \mathbf{X} \rangle \\ \text{subject to} \quad & \mathcal{A}\mathbf{X} = \mathbf{b} \\ & \mathbf{X} \in \mathbb{S}_+^n, \end{aligned}$$

where we do linear optimization subject to affine constraints over the cone of positive-semidefinite matrices. The solver implements the dual interior point method [1] and incorporates several important extensions that enhance the performance of the dual method. HDSDP has been tested over massive datasets and proves a robust and efficient SDP solver.

2 Installation and requirement

This section introduces the steps to install HDSDP and the respective system requirements.

2.1 Access to HDSDP

HDSDP is written in ANSI C and is freely available. Currently HDSDP is maintained at Git repo COPT-Public as part of the open-source project of Cardinal Operations. To install HDSDP, the user can go to

https://github.com/COPT-Public/HDSDP

to download the source code. Or, alternatively, users can execute

```
git clone https://github.com/COPT-Public/HDSDP
```

in the commandline to fetch the source of HDSDP.

2.2 Installation of the SDPA solver

One basic utility offered by HDSDP is solving SDPs in SDPA format (for testing and validation). For this purpose, we provided pre-built binaries in the release section of the Git repo. Users can go to the link

https://github.com/COPT-Public/HDSDP/releases

to download the pre-built binaries. For the detailed instructions on how to use the SDPA solver, please refer to the `Readme.md` file available in the Git repo.

2.3 Installation of the HDSDP library

For more practical uses of HDSDP, it is more common to call HDSDP as a subroutine library. HDSDP adopts CMAKE as its build system. After downloading the repo, the users should see the file structure as follows.

```
HDSDP
├── CMakeLists.txt  # cmake file
├── doc  # documents
├── examples  # examples of use
├── externals  # external libraries
├── include  # headers
├── interface  # HDSDP interfae
├── lib  # linear algebra library
└── src  # source of the algorithm.
```

Since the linear algebraic computation module of HDSDP adopts several sparse linear system routines from the Intel Math Kernal Library, we provide pre-built dynamic libraries named `hdsdplinsys` that already extracted related utilities and the users can also access them from https://github.com/COPT-Public/HDSDP. The downloaded dynamic library should be placed in the `lib` directory.

After checking the availability of the libraries and the sources. The user can move on to modifying the CMakeList.txt file, which reads

```
cmake_minimum_required(VERSION 3.21)
set(CMAKE_OSX_ARCHITECTURES x86_64)
project(hdsdp C)

# Configure paths
set(ENV{HDSDP_HOME} /HDSDP/gwz/cmake)

# Include project directory
include_directories(externals)
...
target_link_libraries(hdsdp $ENV{HDSDP_HOME}/lib/libhdsdplinsys.so)
target_link_libraries(hdsdp m)
```

Here are some steps to complete

1. Ensure that version of `cmake` is above 3.21

2. Modify the variable `HDSDP_HOME` to the path where HDSDP is located.

After finishing the above configurations. The user can execute

```
mkdir build
cd build
cmake ..
make
```

to finish building HDSDP. If successful, the following information would be printed by CMake to the screen

```
build % make
[  1%] Building C object CMakeFiles/hdsdp.dir/externals/config.c.o
[  3%] Building C object CMakeFiles/hdsdp.dir/externals/cs.c.o
[  5%] Building C object CMakeFiles/hdsdp.dir/interface/dsdphsd.c.o
[ 44%] Building C object CMakeFiles/hdsdp.dir/src/dsalg/stepheur.c.o
[ 46%] Building C object CMakeFiles/hdsdp.dir/src/dsalg/symschur.c.o
[ 48%] Building C object CMakeFiles/hdsdp.dir/src/presolve/dsdppresolve.c.o
[ 50%] Linking C executable hdsdp
[ 50%] Built target hdsdp
[ 51%] Building C object CMakeFiles/hdsdplib.dir/externals/config.c.o
[ 53%] Building C object CMakeFiles/hdsdplib.dir/externals/cs.c.o
[ 55%] Building C object CMakeFiles/hdsdplib.dir/interface/dsdphsd.c.o
. . .
[100%] Linking C shared library libhdsdplib.dylib
```

and both an SDPA solver and a dynamic library for HDSDP will be generated in the `build` directory.

3 Solver interface and data input

HDSDP is designed as a stand-alone optimization solver and provides a self-contained interface. After the data is input, the solver will initiate the subsequent solution phases automatically until the solution procedure ends or fails. With the compiled library, the user can call the solver by including the header `dsdphsd.h`.

```
#include "dsdphsd.h"
```

and linking the dynamic library with the applications. The header `dsdphsd.h` defines several utility macros, including status code parameters, constants and most importantly, the solver interface. Before getting down to the solver's interface, we first need to make it clear how HDSDP defines data inputs.

3.1 Data input

Recall that the standard SDP problem is presented by

$$\begin{array}{rll} \min\limits_{\{\mathbf{X}_j\},\mathbf{x}} & \sum\limits_{j=1}^{p} \langle \mathbf{C}_j, \mathbf{X}_j \rangle + \langle \mathbf{c}, \mathbf{x} \rangle & \\ \text{subject to} & \sum\limits_{j=1}^{p} \langle \mathbf{A}_{ij}, \mathbf{X}_j \rangle + \langle \mathbf{a}_i, \mathbf{x} \rangle = b_i, & i = 1, \ldots, m \\ & \mathbf{X}_j \in \mathbb{S}_+^{n_j}, & j = 1, \ldots, p \\ & \mathbf{x} \in \mathbb{R}_+^{n_l} & \end{array}$$

where we introduce multiple $\{\mathbf{X}_j\}$ in case $\mathbf{X} = \begin{pmatrix} \mathbf{X}_1 & & \\ & \ddots & \\ & & \mathbf{X}_p \end{pmatrix}$ exhibits block-diagonal structure and LP variables $\mathbf{x}$ in case some SDP blocks are diagonal. Given the above formulation, we define

- m to be the number of constraints
- p to be the number of blocks
- n_j to be the dimension of the SDP block
- $n := \sum_{j=1}^{p} n_j$ to be the total SDP dimension
- n_l to be the LP dimension.

Given SDP coefficients $\{\mathbf{A}_{i,j}\}$, $\{\mathbf{C}_j\}$; LP coefficients $\{\mathbf{a}_i\}$ and $\mathbf{c}$ and constraint RHS vector $\mathbf{b}$, the SDP is uniquely defined.

3.1.1 SDP data

In HDSDP, the matrix coefficients are presented block-wise and objectives are absorbed into the block. i.e., for each block ID j, we consider $\{\mathbf{A}_{i,j}\}, \mathbf{C}_j$ jointly. For clarity, we drop the j index and assume that we focus on a certain block.

Given a symmetric matrix $\mathbf{A} \in \mathbb{S}^n$,

$$\mathbf{A} = \begin{pmatrix} \downarrow a_{11} & & & \\ \downarrow a_{21} & \downarrow a_{22} & & \\ \vdots & & \ddots & \\ \downarrow a_{n1} & \downarrow a_{n2} & \cdots & \downarrow a_{nn} \end{pmatrix},$$

we define its vectorization $\operatorname{vec}(\mathbf{A})$ by a $\mathbb{R}^{\frac{n\times(n+1)}{2}}$ vector

$$\operatorname{vec}(\mathbf{A}) := \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} & a_{22} & a_{33} & \cdots & a_{nn} \end{pmatrix}^\top,$$

which only contains the lower-triangular part of $\mathbf{A}$ due to symmetry. Then given $m+1$ matrices $\{\{\mathbf{A}_i\}, \mathbf{C}\}$ of equal size, we can vectorize and concatenate them by

$$\mathbf{V} := \begin{pmatrix} | & | & & | & | \\ \operatorname{vec}(\mathbf{A}_1) & \operatorname{vec}(\mathbf{A}_2) & \cdots & \operatorname{vec}(\mathbf{A}_m) & \operatorname{vec}(\mathbf{C}) \\ | & | & & | & | \end{pmatrix} \in \mathbb{R}^{\left[\frac{n\times(n+1)}{2}\right]\times(m+1)}.$$

Last we compress $\mathbf{V}$ using the standard `CSC` and each $\mathbf{A}_i$ is a column of the `CSC` representation.

Example 1. (CSC format of a set of matrices)

Consider the following SDP

$$\begin{aligned} \min_{\mathbf{X}} \quad & \left\langle \begin{pmatrix} 6 & 2 \\ 2 & 3 \end{pmatrix}, \mathbf{X} \right\rangle \\ \text{subject to} \quad & \left\langle \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \mathbf{X} \right\rangle = 1 \\ & \left\langle \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \mathbf{X} \right\rangle = 2 \\ & \mathbf{X} \succeq \mathbf{0}, \end{aligned}$$

which is an SDP of 2 constraints, 1 SDP block, 2 SDP dimension and 0 LP dimension. Then $\mathbf{A}_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \mathbf{A}_2 = \begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix}, \mathbf{C} = \begin{pmatrix} 6 & 2 \\ 2 & 3 \end{pmatrix}$ and the coefficient of block 1 (out of 1) can be expressed by

$$\mathbf{V} = \begin{pmatrix} 1 & 3 & 0 \\ 0 & 1 & 1 \\ 1 & 2 & 0 \end{pmatrix},$$

which, after CSC compression, gives

```
Vp[4] = {0, 2, 5, 6};
Vi[6] = {0, 2, 0, 1, 2, 1};
Vx[6] = {1.0, 1.0, 3.0, 1.0, 2.0, 1.0};
```

3.1.2 LP data

LP data is rather straight-forward. By concatenating

$$\mathbf{A}_l = \begin{pmatrix} — & \mathbf{a}_1 & — \\ & \vdots & \\ — & \mathbf{a}_m & — \end{pmatrix}$$

we compress $\mathbf{A}_l$ to represent the LP data. $\mathbf{c}$ for LP is expressed separately.

3.2 Solver interface

In this section, we give a detailed introduction on the solver interface. The routines are documented in the same order as how they are expected to be invoked.

3.2.1 Creating the solver

Routine

```
extern DSDP_INT DSDPCreate( HSDSolver **dsdpSolver, char *modelName );
```

Explanation

Create the pointer of the solver and initialize

Parameters

- `dsdpSolver`

  Type: `HDSDPSolver **`

  Address of the pointer to which the solver structure is to be allocated

- `modelName`

  Type: `char *`

  Name of the optimization model. If the entry is `NULL`, the model name is left empty

Return

- `DSDP_RETCODE_OK`

  If the input is valid and memory is successfully allocated

- `DSDP_RETCODE_FAILED`

  Otherwise

Example

```
HSDSolver *dsdp = NULL;
retcode = DSDPCreate(&dsdp, NULL);
```

3.2.2 Setting dimension

Routine

```
extern DSDP_INT DSDPSetDim( HSDSolver *dsdpSolver,
                            DSDP_INT  nVars,
                            DSDP_INT  nBlock,
                            DSDP_INT  nConstrs,
                            DSDP_INT  lpDim,
                            DSDP_INT  *nNzs )
```

Explanation

Set the dimension of the SDP to be solved. After this routine is invoked, all the internal working memory is allocated based on the input size provided by the user.

Parameters

- `dsdpSolver`

  Type: `HDSDPSolver **`

  Address of the HDSDP solver

- `nBlock`

  Type: `DSDP_INT`

  Range: ≥1

  Number of SDP blocks

- `nConstrs`

  Type: `DSDP_INT`

  Range: ≥1

  Number of SDP constraints

- `lpDim`

  Type: `DSDP_INT`

  Range: ≥0

  Number of LP variables

- `nNzs`

  Type: `DSDP_INT *`

  Address of the variable indicating number of nonzeros in the coefficient data. Currently only for printing message to the screen and accepts `NULL` is not necessary

Return

- `DSDP_RETCODE_OK`

  If the input is valid and memory is successfully allocated

- `DSDP_RETCODE_FAILED`

  Otherwise

Example

```
retcode = DSDPSetDim(dsdp, MATRIX_DIM, 1, NUM_CONSTR, NUM_LPVAR, NULL);
```

3.2.3 Setting LP data

Routine

```
extern DSDP_INT DSDPSetLPData( HSDSolver *dsdpSolver,
                               DSDP_INT  nCol,
                               DSDP_INT  *Ap,
                               DSDP_INT  *Ai,
                               double    *Ax,
                               double    *lpObj )
```

Explanation

Set the LP data. Note that the number of variables/constraints have to agree with what was set in `DSDPSetDim`.

Parameters

- `dsdpSolver`

Type: `HDSDPSolver *`

Address of the HDSDP solver

- `nCol`

  Type: `DSDP_INT`

  Range: equal to what was set in `DSDPSetDim`

  Number of LP variables. n_l in the introduction part

- `Ap`

  Type: `DSDP_INT *`

  Column pointer of `CSC` representation of LP coefficient matrix $\mathbf{A}_l$. Starting from 0 and of length $n_l + 1$. `Ap[n + 1]` is the number of nonzeros in $\mathbf{A}_l$

- `Ai`

  Type: `DSDP_INT *`

  Row indices of `CSC` representation of LP coefficient matrix $\mathbf{A}_l$. Length is indicated by `Ap[n + 1]`

- `Ax`

  Type: `double *`

  Data of `CSC` representation of LP coefficient matrix $\mathbf{A}_l$. Length is indicated by `Ap[n + 1]`

- `c`

  Type: `double *`

  LP objective vector $\mathbf{c}$

Return

- `DSDP_RETCODE_OK`

  If the input is valid and memory is successfully allocated

- `DSDP_RETCODE_FAILED`

  Otherwise

Example

```
retcode = DSDPSetLPData(dsdp, NUM_LPVAR, Alp, Ali, Alx, lpObj);
```

3.2.4 Setting SDP data

Routine

```
extern DSDP_INT DSDPSetSDPConeData( HSDSolver *dsdpSolver,
                                    DSDP_INT  blockid,
                                    DSDP_INT  coneSize,
                                    DSDP_INT  *Asdpp,
                                    DSDP_INT  *Asdpi,
                                    double    *Asdpx );
```

Explanation

Set the SDP data for some block. Note that the SDP dimension has to agree with what was set in `DSDPSetDim`.

Parameters

- `dsdpSolver`

  Type: `HDSDPSolver *`

  Address of the HDSDP solver

- `blockid`

  Type: `DSDP_INT`

  Range: $<$ Number of blocks set from `DSDPSetDim`

  The index j of the block, not allowed to set the same block twice

- `coneSize`

  Type: `DSDP_INT`

  Range: ≥ 1

  Dimension of the current SDP block

- `Asdpp`

  Type: `DSDP_INT *`

  Row indices of `CSC` representation of SDP coefficient $\mathbf{V}_j$.

- `Asdpi`

  Type: `DSDP_INT *`

  Row indices of `CSC` representation of SDP coefficient $\mathbf{V}_j$.

- `Asdp`

  Type: `double *`

  Data of `CSC` representation of LP coefficient matrix $\mathbf{V}_j$.

Return

- `DSDP_RETCODE_OK`

  If the input is valid and memory is successfully allocated

- `DSDP_RETCODE_FAILED`

  Otherwise

Example

```
retcode = DSDPSetSDPConeData(dsdp, 0, MATRIX_DIM, Asp, Asi, Asx);
```

3.2.5 Setting right-hand-side

Routine

```
extern DSDP_INT DSDPSetObj   ( HSDSolver *dsdpSolver, double *dObj );
```

Explanation

Set the RHS vector **b**.

Parameters

- `dsdpSolver`

  Type: `HDSDPSolver *`

  Address of the HDSDP solver

- `dObj`

  Type: `double *`

  Array containing the RHS vector **b**. Can take `NULL` if $\mathbf{b} = \mathbf{0}$.

Return

- `DSDP_RETCODE_OK`

  If the input is valid and memory is successfully allocated

- `DSDP_RETCODE_FAILED`

  Otherwise

3.2.6 Parameter interface

Routine

```
extern void DSDPSetDblParam ( HSDSolver  *dsdpSolver,
                              DSDP_INT    pName,
                              double      dblVal );
extern void DSDPSetIntParam  ( HSDSolver *dsdpSolver,
                               DSDP_INT   pName,
                               DSDP_INT   intVal );
extern void DSDGetDblParam ( HSDSolver  *dsdpSolver,
                             DSDP_INT   pName,
                             double    *dblVal );
extern void DSDPGetIntParam  ( HSDSolver *dsdpSolver,
                               DSDP_INT   pName,
                               DSDP_INT  *intVal );
```

Explanation

Set or get algorithm parameter for HDSDP.

Parameters

- `dsdpSolver`

  Type: `HDSDPSolver *`

  Address of the HDSDP solver

- `pName`

  Type: `DSDP_INT`

Index of the HDSDP parameter

- `intVal/dblVal`

  Type: `DSDP_INT or double (DSDP_INT * or double *)`

  Value of the parameter to set or to get

Return

- No return

3.2.7 Optimization

Routine

```
extern DSDP_INT DSDPOptimize ( HSDSolver *dsdpSolver );
```

Explanation

Initiate the optimization procedure of dual method.

Parameters

- `dsdpSolver`

  Type: `HDSDPSolver *`

  Address of the HDSDP solver

Return

- `DSDP_RETCODE_OK`

  If the solution is done successfully

- `DSDP_RETCODE_FAILED`

  Otherwise

3.2.8 Get solution

Routine

```
extern DSDP_INT DSDPGetDual   ( HSDSolver *dsdpSolver, double *y, double **S );
extern DSDP_INT DSDPGetPrimal( HSDSolver *dsdpSolver, double **X );
```

Explanation

Get solution after optimization phase ends

Parameters

- `dsdpSolver`

  Type: `HDSDPSolver *`

  Address of the HDSDP solver

- `y`

  Type: `double *`

The array to hold the dual solution

- `S`

  Type: `double **`

  The array of pointers of the same length as the number of SDP blocks. Each pointer should be a $n_j \times n_j$ dense array. On exit, each array is filled by the SDP dual slack.

- `X`

  Type: `double **`

  The array of pointers of the same length as the number of SDP blocks. Each pointer should be a $n_j \times n_j$ dense array. On exit, each array is filled by the SDP primal solution.

Return

- `DSDP_RETCODE_OK`

  If solution is successfully extracted

- `DSDP_RETCODE_FAILED`

  Otherwise

3.2.9 Free the solver

Routine

```
extern DSDP_INT DSDPDestroy  ( HSDSolver *dsdpSolver )
```

Explanation

Free all the internal memories allocated by HDSDP.

Parameters

- `dsdpSolver`

  Type: `HDSDPSolver *`

  Address of the HDSDP solver

Return

- `DSDP_RETCODE_OK`

  If no exception is captured during memory de-allocation.

- `DSDP_RETCODE_FAILED`

  Otherwise

4 Parameters

To get the best performance, it is often necessary to tune the solver parameters. As the last section suggests, HDSDP provides a parameter interface that allows the users to adjust the solver parameters.

4.1 Integer parameters

4.1.1 INT_PARAM_ACORRECTOR

Explanation

The number of corrector steps performed in Phase A. Higher value reduces the iteration number of Phase A but slow downs each iteration. It is suggested increasing the value if $n \ll m$.

Range: ≥ 1

Default: Adjusted by problem feature

4.1.2 INT_PARAM_BCORRECTOR

Explanation

The number of corrector steps performed in Phase B. Higher value accelerates convergence of Phase B but slow downs each iteration. It is suggested increasing the value if $n \ll m$. Also, as Benson suggests [1], the time spent in corrector should not be too long.

Range: ≥ 1

Default: Adjusted by problem feature

4.1.3 INT_PARAM_CGREUSE

Explanation

The parameter controlling behavior of ADPCG solver. Currently not playing a role but will be linked with ADPCG to guarantee deterministic behaviour of HDSDP.

Range: ≥ -1

Default: Adjusted by problem feature

4.1.4 INT_PARAM_AMAXITER (INT_PARAM_BMAXITER)

Explanation

Number of iterations for Phase A and B. Note that dual method often takes more iterations than the primal-dual interior point methods and it is suggested setting time limit instead.

Range: ≥ 1

Default: 500 for each

4.2 Double parameters

4.2.1 DBL_PARAM_RHO(N)

Explanation

Parameter of potential function. Increasing this parameter sometimes speeds up convergence but there is risk of losing track of central path.

Range: ≥ 1.0 and ≤ 10.0

Default: Adjusted by problem feature

4.2.2 DBL_PARAM_POBJ

Explanation

The initial upperbound for the potential function. If an upperbound on primal objective is known, this value can be replaced.

Range: Any

Default: Adjusted by problem feature

4.2.3 DBL_PARAM_INIT_BETA

Explanation

Start dual scaling from $\mathbf{S} = \beta \cdot \|\mathbf{C}\|_F$. Larger values allows more aggressive steps but may result in numerical difficulties. Generally ≥ 1 to ensure an interior initial point.

Range: >0

Default: Adjusted by problem feature

4.2.4 DBL_PARAM_ABS_OPTTOL (DBL_PARAM_REL_OPTTOL)

Explanation

Absolute/relative optimality tolerance. The algorithm stops if

$$\frac{\langle \mathbf{C}, \mathbf{X} \rangle - \mathbf{b}^\top \mathbf{y}}{|\langle \mathbf{C}, \mathbf{X} \rangle| + |\mathbf{b}^\top \mathbf{y}| + 1} \leq \varepsilon_{\text{rel}} \quad \text{and} \quad |\langle \mathbf{C}, \mathbf{X} \rangle - \mathbf{b}^\top \mathbf{y}| \leq \varepsilon_{\text{abs}}.$$

Range: >0

Default: 10^{-6}

4.2.5 DBL_PARAM_PRLX_PENALTY

Explanation

Penalty of primal infeasibility (bound on dual variables). Larger range allows larger range for dual solutions but resuts in numerical difficulties.

Range: >0

Default: Adjusted by problem feature

4.2.6 DBL_PARAM_TIME_LIMIT

Explanation

Solution time limit in CPU seconds. Note that the time limit limits the total solution time, which is counted after Phase A starts.

Range: >0

Default: 15000 seconds.

5 Logging and examples

HDSDP provides a complete logging system that allows the user to monitor different phases of the algorithm. In this section, we dig deeper into the logging system of HDSDP.

5.1 Pre-solving

Before the solution starts, HDSDP spends time on several tasks that may greatly improve the iteration of the dual method.

```
-----------------------------------------------------------------------------------------------
| Homogeneous Dual Scaling Interior Point Solver. Version 0.9.4 (Build date 9.2 2022)
-----------------------------------------------------------------------------------------------
| Reading data from mcp100.dat-s
-----------------------------------------------------------------------------------------------
| nSDPBlock: 1 | nConstrs: 100 | LP. Dim: 0 | SDP. Dim: 100 | Nonzeros: 469
| Data read into solver. Elapsed Time: 0.005 seconds.
-----------------------------------------------------------------------------------------------
| Start presolving
| - Rank one detection completes in 0.000 seconds       <= Detecting low-rank structure
| - Eigen decomposition completes in 0.000 seconds      <= Running SPEIGS
| - Matrix statistics ready in 0.000 seconds            <= Collecting matrix statistics
|    Schur Re-ordering: M1: 0 M2: 100 M3: 0 M4: 1 M5: 0 <= Re-ordering the Schur matrix
| - Schur matrix re-ordering completes in 0.000 seconds    and assign the MX techniques to rows
| - Scaling completes in 0.270 seconds                  <= Matrix scaling
| - Dual symbolic check completes in 0.000 seconds      <= Symbolic analysis of the dual matrix
| - Detecting special structures
| - Special structures found                            <= Detect special structures
|    tr(X) = 1.00e+02 : Bound of X fixed                <= Adjust parameters by structures
| - Special structure detection completes in 0.000 seconds
| Presolve Ends. Elapsed Time: 0.271 seconds
-----------------------------------------------------------------------------------------------
| Matrix statistics [Including C]:                      <= Printing matrix statistics
-----------------------------------------------------------------------------------------------
|       Zero |      Sparse |       Dense |      Rank-1 |      |A|      |      |b|      |      |C|
---------------------------------------------------|-------------------------------------------
|          0 |           1 |           0 |         100 |  1.00000e+02 |  1.00000e+02 |  2.69000e+02
-----------------------------------------------------------------------------------------------
| Parameter Summary:                                    <= Summarize parameters
-----------------------------------------------------------------------------------------------
| Rhon [1.0, 10.0]: 5
| Golden linesearch {0, 1}: 0
| Primal relaxation penalty (0.0, inf): 1e+07
| Time limit (0.0, inf): 15000
| Corrector A: 4  Corrector B: 0
-----------------------------------------------------------------------------------------------
| DSDP is initialized with Ry = -1.000e+05 * I          <= Initialization
| DSDP Phase A starts. Eliminating dual infeasibility
-----------------------------------------------------------------------------------------------
```

The most important role of pre-solving phase is to reveal special structures available by taking a closer look at the problem structure. HDSDP exploits structures including low-rankness, sparsity, implicit (explicit) bounded, homogeniety and so forth. There is a module in HDSDP that is devoted to collecting such structure information and changing the problem parameters based on such information. The purpose of logging here is to reveal such structures to the user, which we believe is not common for SDP solvers.

5.2 Solution logging

After the optimization starts, HDSDP prints iteration log as all the interior point solvers do.

```
Phase A Log: 'P': Primal solution found. '*': Phase A ends. 'F': Error happens. 'M': Max iteration
---------------------------------------------------------------------------------------------------
| Iter |          pObj |          dObj |      dInf |       k/t |        mu |   step |     Pnorm |   E |
---------------------------------------------------------------------------------------------------
|    1 |  1.00000e+05 |  0.00000e+00 |  1.00e+06 | 0.00e+00 | 3.37e+04 |   0.00 |  1.0e+30 |     |
|    2 |  6.73320e+06 | -1.99959e+07 |  0.00e+00 | 0.00e+00 | 3.20e+04 |   1.00 |  2.0e+01 |   * |
---------------------------------------------------------------------------------------------------
Phase A Log Ends.
```

We use the last column `E` to inform the user what is probably happening to the solver. Also, the maintainers can use such information to debug the solver.

5.3 Event profiling

Currently HDSDP uses a quite simple event logging system compared to DSDP5.8.

```
---------------------------------------------------------------------------------------------------
| DSDP Time Summary:
---------------------------------------------------------------------------------------------------
|          Event |    Time(s) |
---------------------------------------------------------------------------------------------------
|       Presolve |      0.271 |
|  Phase A (iter) |      0.021 | (2)
|  Phase B (iter) |      0.017 | (21)
|          Get X |      0.001 |
|      Postsolve |      0.000 |
|            All |      0.310 | (23)
---------------------------------------------------------------------------------------------------
```

A more delicate event profiling system is now under development and will be added to HDSDP.

6 Derivatives

HDSDP combines several heuristics/tricks that prove efficient in practice and they deserve separate attention for the interior point method. To let them go beyond the solver, two additional open-source projected are released as the by-product of HDSDP.

- `SPEIGS`

  An efficient preprocessor for VERY SParse EIgen-decomposition problem.

  Available at https://github.com/leavesgrp/SPEIGS

  This package originates from the pre-solving module of HDSDP, where the sparse SDP coefficient matrices are factorized quite efficiently.

- ADPCG

  An Adaptive Pre-conditioned Conjugate Gradient method

  Available at https://github.com/leavesgrp/ADPCG

  This package implements an abstract pre-conditioned conjugate gradient method with restart. One of its special feature is its internal adaptive mechanism that controls when to re-use the pre-conditioner.

We refer the interested users to the above two derivative projects. They are documented using Doxygen system and also provide detailed examples of use.

7 History of DSDP family

HDSDP is a successor of Benson's DSDP family, which stands the test of time over last 20 years and proves a state-of-the-art implementation of the dual-scaling algorithm. DSDP was initially released by Benson 25 years ago and was consistently maintained and improved till 2006. Its outperformed several primal-dual solvers at the time with its careful and delicate implementation.

15 years later, in 2021, HDSDP was initiated by the authors and aims to explore if dual algorithm has more potential to explore. And after one and a half years, the initial version of HDSDP comes into being. Although HDSDP is implementated independent of DSDP, as we noted in our paper, we are inspired by several ideas from the DSDP developers [1]. Again, we show our respect to the DSDP developers.

Bibliography

[1] Steven J Benson and Yinyu Ye. Algorithm 875: dsdp5—software for semidefinite programming. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–20, 2008.