Cardinal Optimizer (COPT) User Guide

Ver 6.5.3

Cardinal Operations

D. Ge, Q. Huangfu, Z. Wang, J. Wu and Y. Ye

Apr 28, 2023

Contents

Chapter 1

Introduction to Cardinal Optimizer

Cardinal Optimizer is a high-performance mathematical programming solver for efficiently solving large-scale optimization problem. This documentation provides basic introduction to the Cardinal Optimizer, including:

- *How to install Cardinal Optimizer*
- *How to setup license files*
- *How to use Cardinal Optimizer in interactive shell*

We suggest that all users to read the first two sections carefully before using the Cardinal Optimizer.

Once the installation and license setup are done, we recommend users who want to do a quick experiment on the Cardinal Optimizer to read the *COPT Interactive Shell* chapter for details. Users who already have preferred programming language can select from available `Application Programming Interfaces (APIs)`, including:

- *C interface*
- *C++ interface*
- *C# interface*
- *Java interface*
- *Python interface*
- *AMPL interface*
- *Pyomo interface*
- *PuLP interface*
- *CVXPY interface*

1.1 Overview

Cardinal Optimizer supports solving Linear Programming (LP) problems, Second-Order-Cone Programming (SOCP) problems, Convex Quadratic Programming (QP) problems, Convex Quadratically Constrained Programming (QCP) problems, Semidefinite Programming problems (SDP) and Mixed Integer Programming (MIP) problems, which include Mixed Integer Linear Programming (MILP), Mixed Integer Second-Order-Cone Programming (MISOCP), Mixed Integer Convex Quadratic Programming (MIQP), Mixed Integer Convex Quadratically Constrained Programming (MIQCP).

We will support more problem types in the future. The supported problem types and available algorithms are summarized in Table 1.1

Table 1.1: Supported problem types and available algorithms

Problem type	**Available algorithms**
Linear Programming (LP)	Dual simplex; Barrier
Second-Order-Cone Programming (SOCP)	Barrier
Convex Quadratic Programming (QP)	Barrier
Convex Quadratically Constrained Programming (QCP)	Barrier
Semidefinite Programming (SDP)	Barrier; ADMM
Mixed Integer Linear Programming (MILP)	Branch-and-Cut
Mixed Integer Second-Order-Cone Programming (MISOCP)	Branch-and-Cut
Mixed Integer Convex Quadratic Programming (MIQP)	Branch-and-Cut
Mixed Integer Convex Quadratically Constrained Programming (MIQCP)	Branch-and-Cut

Cardinal Optimizer supports all major 64-bit operating systems including Windows, Linux (including ARM64 platform) and MacOS (including ARM64 platform), and currently provides programming interfaces shown below:

- C interface
- C++ interface
- C# interface
- Java interface
- Python interface
- AMPL interface
- Pyomo interface
- PuLP interface
- CVXPY interface
- GAMS interface
- Julia interface

We are going to develop more programming interfaces to suit various needs of users and situations.

1.2 Licenses

Now, we provides 4 types of license, which are Personal License, Server License, Floating License, and Cluster License. They are listed in table below (Table 1.2) :

Table 1.2: License Type

License Type	Detail
Personal License	It is tied to personal computers by username. Only approved user can run COPT on his devices. No limitations on CPU cores and threads.
Server License	It is tied to a single server computer by its hardware info (MAC and CPUID). An arbitrary number of users and programs can run COPT simultaneously. No limitations on CPU cores as well.
Floating License	It is tied to a server machine running COPT floating token service, by its hardware info (MAC and CPUID). Any COPT floating client connected to server can borrow and use the floating license, thus run one process for optimization jobs simultaneously. The token number is max number of clients who can use floating licenses simultaneously.
Cluster License	It is tied to a server machine running COPT compute cluster service, by its hardware info (MAC and CPUID). Any COPT compute cluster client connected to server can offload optimization computations. That is, clients are allowed to do modelling locally, execute optimization jobs remotely, and then obtain results interactively. Although server can have multiple clients connected, each connection must run optimization jobs sequentially. No limitations on CPU cores.

1.3 How to Cite

If you used COPT in your research work, please mention us in your publication. For example:

- We used COPT [1] in our project.
- To solve the integer problem, we used Cardinal Optimizer [1].

with the following entry in the Reference section:

```
[1] D. Ge, Q. Huangfu, Z. Wang, J. Wu and Y. Ye. Cardinal Optimizer (COPT) user guide. https://
↪guide.coap.online/copt/en-doc, 2023.
```

The corresponding BiBTeX citation is:

```
@misc{copt,
  author={Dongdong Ge and Qi Huangfu and Zizhuo Wang and Jian Wu and Yinyu Ye},
  title={Cardinal {O}ptimizer {(COPT)} user guide},
  howpublished={https://guide.coap.online/copt/en-doc},
  year=2023
}
```

1.4 Contact Information

Cardinal Optimizer is developed by Cardinal Operations, users who want any further help can contact us using information provided in Table 1.3

Table 1.3: Contact information

Type	Information	Description
Website	https://www.shanshu.ai/	
Phone	400-680-5680	
Email	coptsales@shanshu.ai	business support
Email	coptsupport@shanshu.ai	technical support

Chapter 2

Installation Guide

This chapter introduces how to install **Cardinal Optimizer** on all supported operating systems, and how to obtain and setup license correctly. We recommend all users ready this chapter carefully before using Cardinal Optimizer.

2.1 Registration

Before using Cardinal Optimizer, users need to registrate online and then install the COPT package on your machine. If this is not done yet, please visit official COPT page and fill the registration form following the guidelines.

The online registration is for personal license application. Specifically, users only need to provide username of machine, besides basic information.

Upon approval, you will receive a letter from coptsales@shanshu.ai. It gives both link to download COPT software package and a license key tied with registration information. You may refer to *Software Installation* below and *Setting Up License* for further steps.

If you encountered any problems, please contact coptsupport@shanshu.ai for help.

2.2 Software Installation

2.2.1 Windows

We provide two types of installation packages for Windows operating systems, one is an executable installer for most of users and the other one is a zip-format archive specialized for expert users. We recommend users to download the excecutable installer.

If you download the executable installer for Windows from our website, e.g. CardinalOptimizer-6.5.2-win64-installer.exe for 64-bit version of COPT 6.5.2, just double-click it and follow the following guidance:

- Step 1: Click the installer and select the installation language. The default installation language is `English`, users can change it by select from the drop-down menu, see Fig. 2.1. Here we use the default setting.

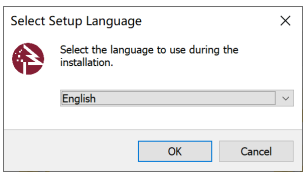


Fig. 2.1: Select installation language

- Step 2: If you agree with the `'End-User License Agreement (EULA)'`, just choose `'I accept the agreement'` and then click `'Next'`. The software won't install if you disagree with the `EULA`, see Fig. 2.2.

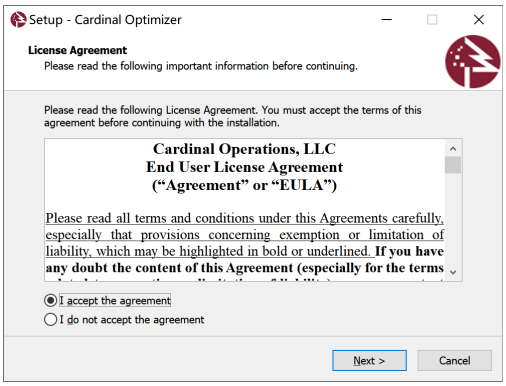


Fig. 2.2: License agreement page

- Step 3: By default, the installer will place all files into directory C:\Program Files\copt65, you may change it to any directories. If you have decided the install directory, just click `'Next'`, see Fig. 2.3.

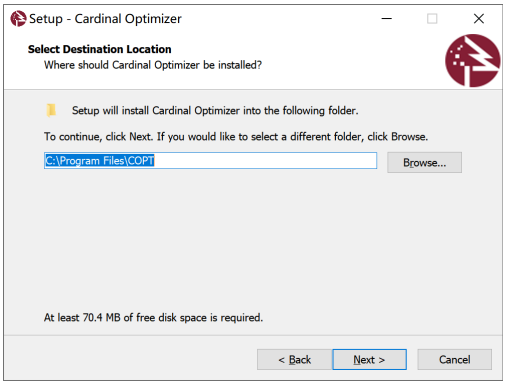


Fig. 2.3: Choose install directory

- Step 4: To select the start menu folder, you can simply use the default setting and click `'Next'`, see Fig. 2.4.

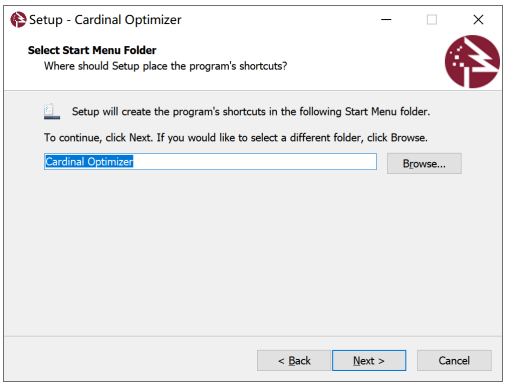


Fig. 2.4: Select start menu folder

- Step 5: By now, the software is ready to install, just click `'Install'`, see Fig. 2.5.

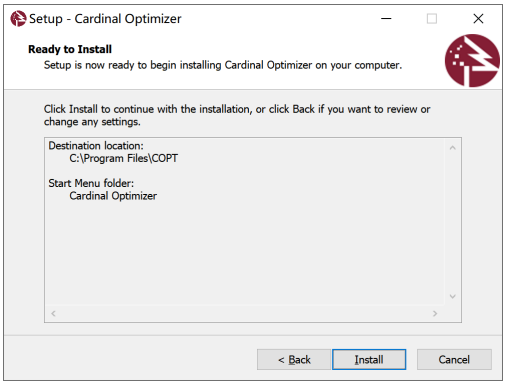


Fig. 2.5: Ready to install

- Step 6: When installation completed, the software requires restart of your machine since the installer has automatically made the required modifications to environment variables for you. Be sure to save your working files and close other running applications before restart, and then click `'Finish'`, see Fig. 2.6.

Fig. 2.6: Installation completed and restart your machine

If you downloaded the zip-format installer archive, just uncompress it to any directories with any unarchiver and set up environment variables as follows. Here we assumed the installation directory to be C:\Program Files\copt65:

- Step 1: Open `Command Prompt (cmd)` with **administrator priviledge** and execute the following command to pop-up the environment variables setting panel.

```
rundll32 sysdm.cpl,EditEnvironmentVariables
```

- Step 2: Add directory C:\Program Files\copt65\bin to system environment variable `PATH`.
- Step 3: Create a new system environment variable named `COPT_HOME`, whose value is C:\Program Files\copt65.
- Step 4: Create a new system environment variable named `COPT_LICENSE_DIR`, whose value is C:\Program Files\copt65.

Up to now, you have already setup the required modifications to the environment variables. If you accept the `copt-eula_en.pdf` in installation directory, then please go to *Setting Up License* for license issues.

2.2.2 MacOS

We provide two types of installation packages for MacOS, one is a DMG-format installer for most of users and the other is a gzip-format archive for expert users. We recommend users to download the DMG-format installer.

DMG installer

If you download the DMG-format installer for MacOS from our website, e.g. CardinalOptimizer-6.5.2-osx64.dmg for 64-bit version of COPT 6.5.2, please follow the following guidance to install the software:

- Step 1: Double-click the DMG-format installer, waiting for the OS to mount the DMG installer automatically.
- Step 2: Simply drag the copt65 folder into `'Applications'` folder, see Fig. 2.7:

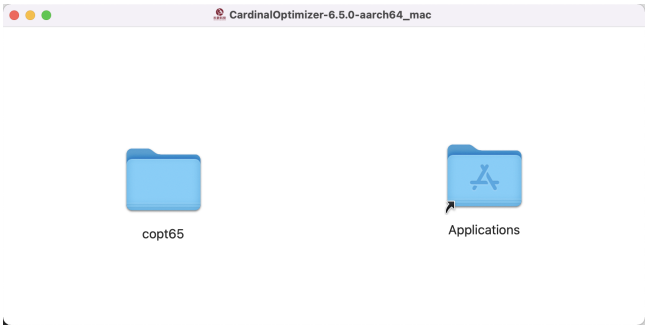


Fig. 2.7: Drag copt65 into `'Applications'`

GZIP installer

If you download the gzip-format installer archive, just uncompress it to any directories with commands:

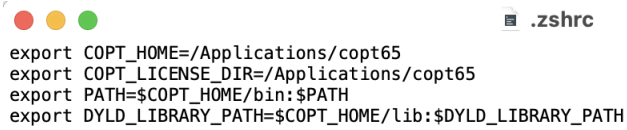
```
tar -xzf CardinalOptimizer-6.5.2-osx64.tar.gz
```

You will get a folder named copt65 in current directory, you can move it to any other directories as you want. We recommend users move it to the `'Applications'` folder by executing commands below:

```
mv copt65 /Applications/
```

BASH terminal

The next step for both DMG-format and gzip-format installer is to setup the environment variables by adding the following commands to the `'.bash_profile'` file in your `$HOME` directory using any editors that you prefered:



Remember to save your modifications to the `'.bash_profile'` file, if it does not exist, you can create at your own by executing:

```
touch ~/.bash_profile
```

ZSH terminal

If your terminal is `zsh`, you should add the following line to `'.zshrc'` file in `$HOME` directory as well:

```
source ~/.zshrc
```

Verify environment variable configuration

Now open a new terminal to check if the previous modifications work by executing commands below respectively:

```
echo $COPT_HOME
echo $COPT_LICENSE_DIR

echo $PATH
echo $DYLD_LIBRARY_PATH
```

If the terminal outputs respectively:

`/Applications/copt65`

`/Applications/copt65`

`/Applications/copt65/bin:$PATH`

`/Applications/copt65/lib:$DYLD_LIBRARY_PATH`

It means that the COPT-related environment variables are configured successfully.

Notes: The environment variables `$PATH` and `$DYLD_LIBRARY_PATH` may display different contents on different computers. However, the set COPT-related environment variables should be displayed normally to indicate that the configuration is successful.

If the user checks that the COPT-related environment variables have been successfully added, please carefully read the user agreement document `'copt-eula_cn.pdf'` in the installation directory. If you accept the the agreement, please go to *Setting Up License* for license issues.

MacOS Security Checkup

For MacOS 10.15 (Catalina), if users received error message below:

```
"libcopt.dylib" cannot be opened because the developer cannot be verified.
macOS cannot verify that this app is free from malware.
```

or

```
"libcopt_cpp.dylib": dlopen(libcopt_cpp.dylib,6): no suitable image found.
Did find: libcopt_cpp.dylib: code signature in (libcopt_cpp.dylib) not valid
for use in process using Library Validation: library load disallowed by
system policy.
```

Then execute the following commands:

```
xattr -d com.apple.quarantine CardinalOptimizer-6.5.2-osx64.dmg
xattr -d com.apple.quarantine CardinalOptimizer-6.5.2-osx64.tar.gz
```

or

```
xattr -dr com.apple.quarantine /Applications/copt65
```

to disable security check of MacOS.

2.2.3 Linux

For Linux platform, we provide gzip-format archive only, if you download the software from our website, e.g. CardinalOptimizer-6.5.2-lnx64.tar.gz for 64-bit version of COPT 6.5.2, just type commands below in shell to extract it to any directories:

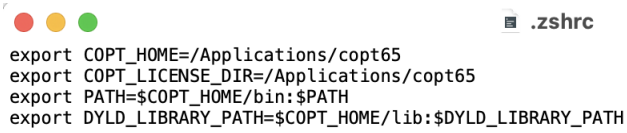
```
tar -xzf CardinalOptimizer-6.5.2-lnx64.tar.gz
```

You will get a folder named copt65 in current directory, you can move it to any other directories as you like. We recommend users move it to `'/opt'` directory by typing commands below in shell:

```
sudo mv copt65 /opt/
```

Note that the above command requires **root privilege** to execute.

The next step users need to do is to set the required environment variables, by adding the following commands to the `'.bashrc'` file in your `$HOME` directory using any editors that you prefered:



Remember to save your modifications to the `'.bashrc'` file, and open a new terminal to check if it works by executing commands below respectively:

```
echo $COPT_HOME
echo $COPT_LICENSE_DIR

echo $PATH
echo $LD_LIBRARY_PATH
```

If you accept the `copt-eula_en.pdf` in installation directory, then please go to *Setting Up License* for license issues.

2.3 Setting Up License

The Cardinal Optimizers requires a valid license to work properly. We offer different types of licenses most suitable for user's needs. All users should read this section carefully. If you encounter any problem about license, feel free to contact coptsupport@shanshu.ai.

Once the registration is done, a license key is sent to users. It is a unique token binding with user's registration information. Afterwards, users may run the `copt_licgen` tool, shipped with Cardinal Optimizer, to obtain license files from COPT licensing server (Internet connection required) .

The following notes show you how to play with the `copt_licgen` tool.

2.3.1 Obtaining License

For Windows system, open a new cmd window. The current path is the user directory, and the path is as follows: `"C:\Users\shanshu"`.

For MacOS and Linux systems, open a new terminal. The current path is the user directory, represented by the symbol `~`.

To obtain COPT license files, execute the following command using the option `'-key'` and the license key as argument. Below is an example, assuming the license key is `'19200817f147gd9f60abc791def047fb'`:

```
copt_licgen -key 19200817f147gd9f60abc791def047fb
```

If the license key is saved to `key.txt` file in format of `'KEY=xxx'`, which resides in the same place as `copt_licgen`, execute the following command using the option `'-file'` and `'key.txt'` as arugment.

```
copt_licgen -file key.txt
```

We recommend to use the **first way** to get your license files.

If the authorization server verifies the license, then it generates `license.dat` and `license.key` and download it to the user's computer. The default download directory is the current working directory.

```
copt_licgen -key 19200817f147gd9f60abc791def047fb
  [Info] Cardinal Optimizer   COPT v6.5.2 20230322
  [Info] Use specific key 19200817f147gd9f60abc791def047fb
  [Info] * get new COPT license from licensing server *
  [Info] Write to license.dat
  [Info] Write to license.key
  [Info] Received new license files from server
  [Info] Done !!!
```

Note: Users do not need to have internet connection to run COPT. However, obtaining license itself requires internet connection. If you encounter any problem, please feel free to contact coptsupport@shanshu.ai.

2.3.2 Verifying License

If the license key binding to registration information is verified by COPT license server, two license files, `license.dat` and `license.key`, are downloaded to the current working directory. To double check two license files are valid in current version of COPT, execute the following command with the option `'-v'`. Note that this command requires both `license.dat` and `license.key` existing in the current working directory.

```
copt_licgen -v
```

If you see log information similar to the following, you have obtained and verified the license files sucessfully.

```
copt_licgen -v
  [Info] Cardinal Optimizer   COPT v6.5.2 20230322
  [Info] Run local validation
  [Info] Read license.dat
  [Info] Read license.key
  [Info] Expiry : Tue 2030-12-31 00:00:00 +0800
  [Info] Local validation result: Succeeded
  [Info] Done !!!
```

2.3.3 Installing License

Once you obtained the `license.dat` and `license.key` files from COPT license server and verified they work as expected, setting up them is just as simple as moving them to the same directory as the COPT dynamic libary or *COPT Command-Line*. Note this installation is applied only to current COPT.

The following two ways of installation applied to all versions of COPT on your machine.

Via HOME directory method (recommended)

The simplest way to install COPT license is to create a folder of `copt` in your `HOME` directory, and move the authenticated license files `license.dat` and `license.key` to the new folder `copt`.

Note: The folder name `"copt"` is case sensitive and it must be lowercase.

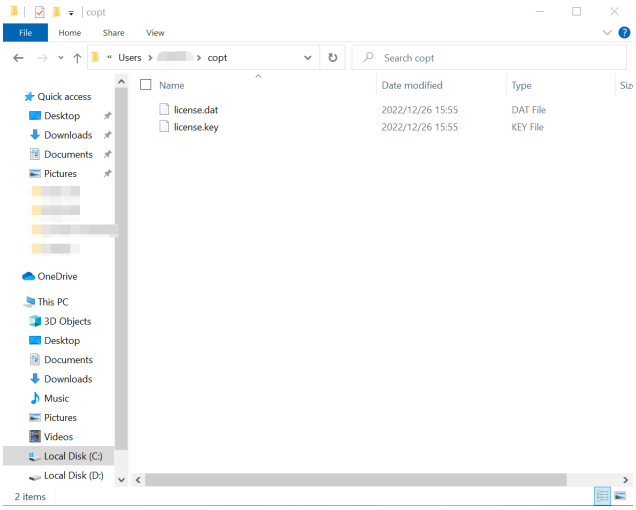
The `HOME` directory names are slightly different under different systems, please refer to the corresponding license installation steps according to your operating system:

- *Windows*
- *MacOS*
- *Linux*

Windows

The `HOME` directory looks like `"C:\Users\username\"` on Windows. User can manually move the two license files `license.dat` and `license.key` to the directory `"C:\Users\username\copt"`.

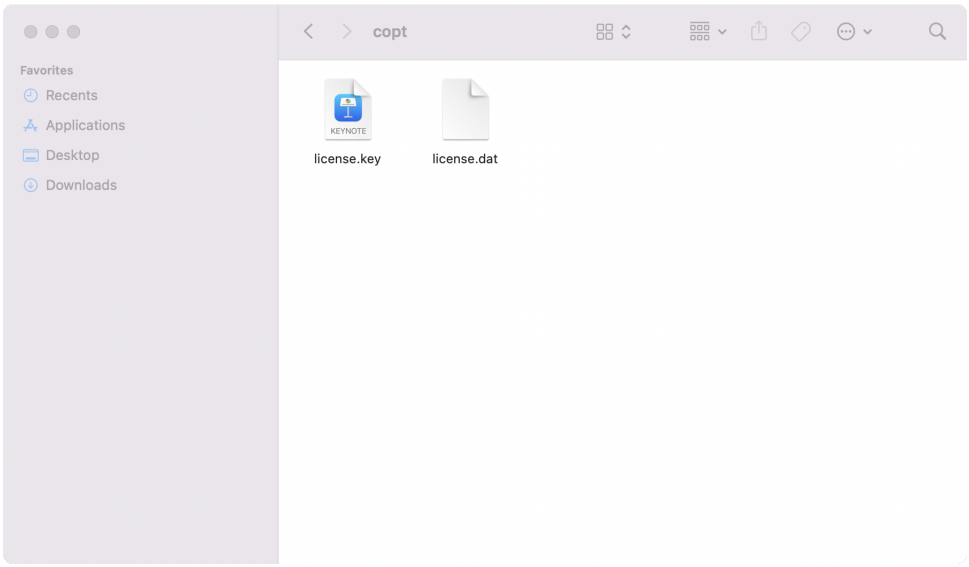
Finally, please check whether the license files `license.dat` and `license.key` are located in this directory, as shown in the figure below, indicating that the license has been installed correctly:



MacOS

The `HOME` directory looks like `"/home/your username/"` on MacOS. User can manually move the two license files to `"/home/your username/copt/"`.

Finally, please check whether the license files `license.dat` and `license.key` are located in this directory, as shown in the figure below, indicating that the license has been installed correctly:



Linux

The `HOME` directory looks like `"/home/your username/"` on Linux.

User can execute the following command in the terminal to move the two license files `license.dat` and `license.key` to the directory `"/home/your username/copt"`.

```
mv license.*  ~/copt/
```

Next, please check that the license files `license.dat` and `license.key` are located in `"/home/username/copt"` directory, the command is:

```
ls ~/copt/
```

If the terminal output shows that there are two files `license.dat` and `license.key`, it means that the license has been installed correctly, otherwise the installation fails.

Via environment variable method

Alternatively, for users who prefer having licenses in a customized folder, they can set environment variable `COPT_LICENSE_DIR` to the customized folder. You may refer to *Software Installation* for how to set environment variable on Windows, Linux and MacOS.

In addition, please double check that if license files `license.dat` and `license.key` locate in path specified by environmental variable `COPT_LICENSE_DIR`.

The viewing and operation of environment variables are slightly different under different systems, please refer to the following installation steps for your own operating system:

- *Windows*
- *MacOS*
- *Linux*

Windows

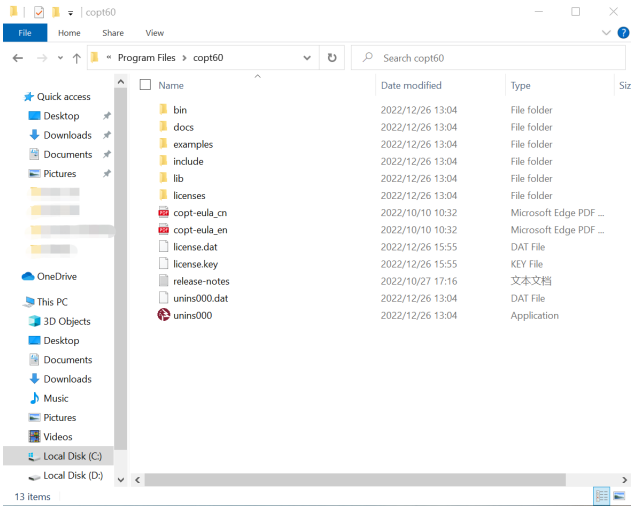
For Windows systems, execute the following command to view the path pointed to by the environment variable `COPT_LICENSE_DIR` :

```
echo %COPT_LICENSE_DIR%
```

Note: If there is no output on the terminal, it means that the COPT-related environment variables have not been configured, please check *Software installation*.

Then move the license files `license.dat` and `license.key` to the path `COPT_LICENSE_DIR` points to.

Here we assume that the environment variable `COPT_LICENSE_DIR` points to the default installation directory of COPT. As shown in the figure, it means that the license has been installed correctly:



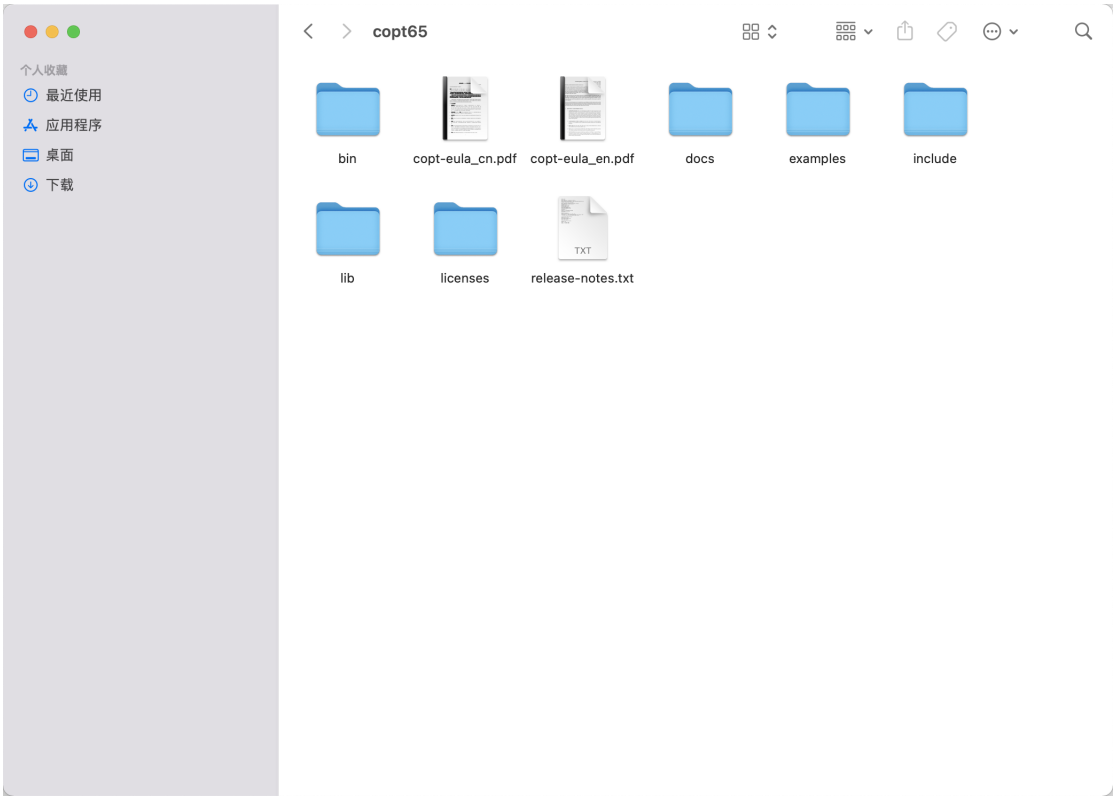
MacOS

For MacOS system, you can use the following command to view the path pointed to by the environment variable `COPT_LICENSE_DIR`:

```
echo $COPT_LICENSE_DIR
```

Then move the license files `license.dat` and `license.key` to the path `COPT_LICENSE_DIR` points to.

Here we assume that the environment variable `COPT_LICENSE_DIR` points to the default installation directory of COPT. As shown in the figure, it means that the license has been installed correctly:



Linux

For Linux systems, user can use the following command to view the path pointed to by the environment variable `COPT_LICENSE_DIR`:

```
echo $COPT_LICENSE_DIR
```

Then user can execute the following command to move the license files `license.dat` and `license.key` to to the path `COPT_LICENSE_DIR` points to.

```
mv license.* $COPT_LICENSE_DIR/
```

Note: For Linux systems, if the path pointed to by the environment variable `COPT_LICENSE_DIR` is /opt/copt65, user need to perform the move operation with **root authority** , and the command is:

```
sudo mv license.* $COPT_LICENSE_DIR/
```

In addition, please double check that if license files `license.dat` and `license.key` locate in path specified by environmental variable `COPT_LICENSE_DIR` , and the command is:

```
ls $COPT_LICENSE_DIR/
```

If the terminal displays the files `license.dat` and `license.key`, it means that the license has been installed correctly, otherwise the license installation fails.

Note: If there are license files `license.dat` and `license.key` in the `copt` folder of the user directory and the directory pointed to by the environment variable, the former will be checked and used the former.

2.3.4 Others

Basically, each type of licenses includes two license files: `license.dat` and `license.key`, each of which has digital signature to protect its content. When invoking COPT command-line or loading COPT dynamic library to solve an optimization problem, the public RSA key stored in `'license.key'` is used to verify signature in `'license.dat'`.

Afterwards, license data in format of key-value pair is parsed to verify whether it is a legal license. Below is a sample of license data.

```
## SHANSHU LICENSE FILE ##

USER = Trial User
MAC = 44:05:99:31:41:C2
CPUID = BFEBFBFF000706E5
EXPIRY = 2030-12-31
VERSION = 6.5.2
NOTE = Free For Trial
```

Chapter 3

COPT Command-Line

The **Cardinal Optimizer** ships with `copt_cmd` executable on all supported platforms, which let users solve optimization models in an interactive way. Before running COPT command-line, please make sure that you have valid license installed.

3.1 Overview

COPT command-line is a COPT API interpreter that executes commands read from the standard input or from a script file. COPT command-line interprets the following options when it is invoked:

- `-c`: If the `'-c'` option is present, it reads from an inline scripts, which is a quoted string and specified by the second argument.
- `-i`: If the `'-i'` option is present, it reads from an input script file, whose path is specified by the second argument.
- `-o`: If the `'-o'` option is present, it reads from standard input, while each **valid** command line is written to an output script file, whose path is specified by the second argument.

Regardless of arguments, the tool is interactive. Besides wrapping COPT API calls, it offers various features to help users move cursor around and edit lines. We try to provide as much user experience as standard command prompt (Windows console and Unix terminal).

3.2 Edit mode

This tool defines a number of commands to position the cursor, edit lines with combination keys on a standard keyboard. The following notes show you how to use the most important ones.

- **Basic commands**
  1. `<Insert>`: Toggle between inserting characters and replacing the existing ones.
  2. `<Esc>`: Discard inputs and move the cursor to the beginning of line. Press `<ESC>` twice on Linux/Mac platform to do the same thing.
- **Moving around**
  1. `<Home>/<End>`: Jump to the beginning/end of line.
  2. `<Left>/<Right>` Arrow: Move the cursor one character to the left/right.
  3. `<CTRL>+<Left>/<Right>` Arrow: Move the cursor one word to the left/right.
- **Cut and Paste**

  An internal paste buffer is available for the following cut operations.

1. <Delete>: Cut the character under the cursor.
2. <Backspace>: Cut the character before the cursor.
3. <CTRL>+<H>: Cut from the cursor to the beginning of line.
4. <CTRL>+<E>: Cut from the cursor to the end of line.
5. <CTRL>+<Y>: Paste text in paste buffer at the cursor position.

Each of cut operations defines cut direction: cut forward or cut backward. Obviously, <Delete> and <CTRL>+<E> cut forward; <Backspace> and <CTRL>+<H> cut backward. When two consecutive cut operations have the same cut direction, the cutting text is appended the paste buffer. Otherwise, the paste buffer is overwritten by the latest chopped text.

- **Command history**

  1. <Up>/<Down>: Move through the history of command lines in the older/newer direction. The tool remembers the history entry if the last executed line is in history.
  2. <CTRL>+<R> or <F8>: If you know what a previously executed line starts with, and you want to run it again, type prefix characters and then press <CTRL>+<R>, or <F8> on Windows platform, to iterate through the history of commands with matching prefix.

- **Tab completion**

  Use <Tab> to complete shell commands, COPT paramters/attributes, or files under specified path. To cycle though multiple matches, just repeat pressing <Tab>.

  1. If the cursor is over or right after the first word on the current command line, press <Tab> to complete available shell commands with matching prefix (from the cursor to the first character of word).
  2. Otherwise, press <Tab> to complete COPT parameters/attributes, or file names under path with matching prefix. Specifically, if the prefix matches with COPT parameters as well as file under current working directory, only COPT parameters will be listed. In this case, to iterate file names, add relative path `'./'` to start with.
  3. For convenience, tab completion ignores case and support asterisk (*) as wildcard to match file and directory pattern.
  4. <Shift>+<Tab>: Complete the next one in an opposite direction.

3.3 Script mode

There are two approaches to run scripts, a batch of commands, in COPT command-line. One is to save scripts as a text file. The other is called **inline scripts**, that is, a quoted string of commands separated by `';'`. Both of them can be loaded when COPT command-line is invoked, or loaded on fly in the edit mode (see shell command `'load'`). Below describes more details about loading scripts as arguments.

This tool allows users to load a script file to do a batch job automatically. As mentioned in *Overview*, a script file is read when its file path is specified as arguments of the option `'-i'`.

- When reading a script file, COPT command-line double checks whether the first non-blank line starts with special text: `'#COPT script file'`. This is to make sure users do not load an invalid script file by mistake. Indeed, only `'#COPT'` is verified. In addition, any line in scripts is commented out if its first non-blank character is `'#'`.
- After a script file is loaded, the tool keeps reading it as standard inputs, until reaching end of file or a special character `'?'`. Here, we use question mark `'?'` to pause scripts on purpose. To continue, users can type `'load'` in command line. Afterwards, the tool picks whatever left in scripts and start to run from there, until reaching end of file or another question mark `'?'`. Once current scripts finish, users can load any other script file on fly.

It also allows users to load special scripts, called **inline scripts**. The only difference from a script file is that commands are separated by `';'`, instead of `'\n'`. So inline scripts can be read by using arguments of the option `'-c'`, or loaded on fly by specifying a quoted string, and special character `'?'` works in the same way.

In addition, this tool provides a feature of recording **valid** command lines sequentially to a script file, if users specify an output script file as argument of the option `'-o'`. Here, **valid** command must use known shell commands and do not exceed number of allowed parameters.

In particular, if users load a script file or inline scripts on fly, all commands in scripts are written to the output script file. Note that command `'load'` itself is not written to output script file on purpose. Because we've expanded and written all commands in scripts. On the other hand, it may trigger infinity loop if the script file loaded is actually itself.

3.4 Shell commands

COPT command-line supports the following shell commands for users to manipulate optimization models. Moreover, shell commands are case-insensitive and support tab-completion.

3.4.1 General shell commands

The shell commands below are in support of interactions.

- `cd`: This shell command works similar to DOS command `'cd'`. That is, it changes **current working directory**, if its argument is valid relative or absolute path of a directory. Note that **current working directory** is the base directory for relative path and tab completion. It is initialized to current binary folder where `copt_cmd` exist. Users can change it by shell command `'cd <dirpath>'`. For example, if users change working directory to a folder having mps files, reading model becomes much easier because only filename is needed.
- `dir/ls`: This shell command works similar to DOS command `'dir'` or Bash command `'ls'`. That is, it lists all files and directories under given relative or absolute path. To see files under current working directory, type `'dir'` or `'ls'`; To see files under parent folder, type `'dir ../'`; To see files under home path, type `'dir ~/'`, etc. In addition, wildcard (`*`) is supported as well. That is, `'dir net'` lists all file names starting with `'net'` under current working directory; `'dir /home/user/*.gz'` lists all files of type of `'.gz'` under path of `'/home/user/'`.
- `exit/quit`: Leave COPT command-line.
- `help`: It provides information on all shell commands. Typing `'help'` followed by a shell command name gives you more details on shell commands. In particular, typing `'help'` without arguments lists all shell commands with short descriptions. Right after overview of shell commands, the text `'help'` with additional whitespace appear in the new prompt line. So users can directly type, or possibly `<Tab>` complete, actual shell command to read more details.
- `load`: Load a script file or inline scripts on fly and then execute a batch of commands. The syntax of `'load'` command should be followed by either relative/absolute path of a script file, or quoted string of inline scripts. One special scenaio is when current script is paused, that is, hit question mark (`'?'`) during execution. In this case, type `'load'` will continue the paused scripts. If users forgot having scripts in progress and try to load another scripts, it works as command `'load'` and any additional argument is ignored. This behavior is back to normal after reaching the end of current scripts.
- `pwd`: This shell command works similar to Bash command `'pwd'`. That is, display current working directory to let users know where they are.
- `status`: COPT command-line has a state machine on status of problem solving (see Fig. 3.1). This is used to guide users through steps. Typing `'status'` shows you current interactive status. The status exposed to users are as follows:
  - `Initial`, initial status, either right after the tool is invoked, or shell command `reset` is called.

– `Read`, read an optimization model in format of mps successfully.

– `SetParam`, set value of any COPT parameter successfully.

– `Optimize`, shell command `opt` is called to solve current optimization model.

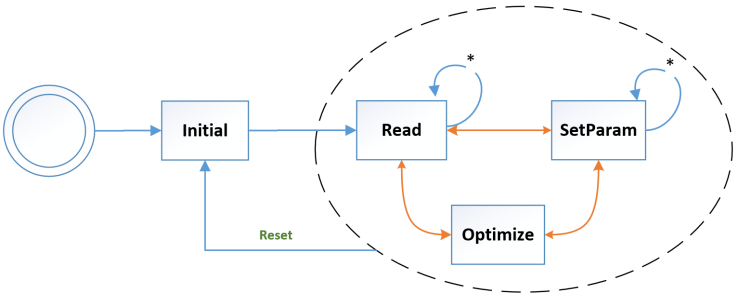


Fig. 3.1: Status of COPT command-line

3.4.2 COPT shell commands

The shell commands below are related to COPT API calls.

- `display/get`: Display current setting of any COPT parameter or attribute. Typing `'display'` followed by COPT parameter or attribute name shows its current value. Typing `'display'` only lists all COPT parameters and attributes with short descriptions. Right after overview of COPT parameters/attributes, the text `'display'` with additional whitespace appear in the new prompt line. So users can directly type, or possibly `<Tab>` complete, actual parameter/attribute name to see its current value.
- `opt/optimize`: Solve optimization model and display results on screen. This command requires no parameter and its interactive status is set to `'Optimize'`.
- `optlp/optimizelp`: Solve optimization model as LP model. This command requires no parameter and its interactive status is set to `'Optimize'`.
- `iis`: Compute IIS for loaded model.
- `feasrelax`: Do feasible relaxation for infeasible problem. Note that an optimization problem must exist before calling 'feasrelax'. `'feasrelax'` or `'feasrelax all'` means to relax all bounds of variables and constraints with penalty 1, while `'feasrelax vars'` means to only relax bounds of variables with penalty 1, and `'feasrelax cons'` means to only relax bounds of constraints with penalty 1.
- `tune`: Tuning parameter automatically of the read-in model.
- `loadtuneparam`: Loads the specified tuning results into the currently read-in model. The calling command is `'loadtuneparam idx'`, where `'idx'` is the specified tuning result's number. Use the command `'get TuneResults'` to get the number of tuning results obtained after the current tuning.
- `read`: Read optimization model, MIP solution, LP basis, MIP initial solution and COPT parametes from file under given relative/absolute path. It supports files including optimization problem file of type `'.mps'` and compressed format `'.mps.gz'`, problem file of type `'.lp'` and compressed format `'.lp.gz'`, problem file of type `'.dat-s'` and compressed format `'.dat-s.gz'`, problem file of type `'.cbf'` and compressed format `'.cbf.gz'`, problem file of type `'.bin'` and compressed format `'.bin.gz'`, basis file of type `'.bas'`, solution file of type `'.sol'`, MIP initial solution file of type `'.mst'`, parameter file of type `'.par'` and tuning file of type `'.tune'`.
- `readmps`: Read optimization model in format of `'.mps'` or `'.mps.gz'` from file. Note this command does not require file type `'.mps'` or `'.mps.gz'`. That is, it is fine to have content in format of MPS, regardless of its file name. In addtion, the interactive status is set to `'Read'`.

- `readlp`: Read optimization model in format of `'.lp'` or `'.lp.gz'` from file. Note this command does not require file type `'.lp'` or `'.lp.gz'`. That is, it is fine to have content in format of LP, regardless of its file name. In addtion, the interactive status is set to `'Read'`.
- `readsdpa`: Read an optimization problem in format of `'.dat-s'` or `'.dat-s.gz'` from file. Note that this command is similar to `'read'` when the file type is `'.dat-s'` . However, `'readsdpa'` does not require the file has type `'.dat-s'`. It parses the file as SDPA format, no matter what file type it is.
- `readcbf`: Read an optimization problem in format of `'.cbf'` or `'.cbf.gz'` from a file. Note that this command is similar to `'read'` when the file type is `'.cbf'` . However, `'readcbf'` does not require the file has type `'.cbf'`. It parses the file as CBF format, no matter what file type it is.
- `readbin`: Read optimization model in format of `'.bin'` or `'.bin.gz'` from file. Note this command does not require file type `'.bin'` . That is, it is fine to have content in format of COPT binary format, regardless of its file name. In addtion, the interactive status is set to `'Read'`.
- `readsol`: Read MIP solution from file. Note this command does not require file type of `'.sol'`. That is, it is fine to have content of solution, regardless of its file name.
- `readbasis`: Read optimal basis from file. Note this command does not require file type of `'.bas'`. That is, it is fine to have content of basis, regardless of its file name.
- `readmst`: Read MIP initial solution from file. Note this command does not require file type of `'.mst'`. That is, it is fine to have content of solution, regardless of its file name.
- `readparam`: Read COPT parameters from file and set corresponding values. Note this command does not require file type of `'.par'`. That is, it is fine to have content of COPT parameters, regardless of its file name.
- `readtune`: Read the parameter file under the given relative or absolute path. Note this command does not require file type of `'.tune'` . That is, it is fine that the file conforms to the COPT parameter tuning file format, regardless of its file name.
- `reset`: Reset current optimization model and all parameters/attributes to defaults. In addtion, the interactive status is set to `'Initial'`.
- `set`: Set value of any COPT parameter. The syntax of this command should be `'set'`, followed by COPT parameter name and an integer/double number. Moreover, Typing `'set'` only lists all COPT parameters with short descriptions. Right after overview of COPT parameters, the text `'set'` with additional whitespace appears in the new prompt line. So users can directly type, or possibly `<Tab>` complete, actual parameter name to complete as partial command. If so, its current value, default value, min value, max value of given COPT parameter are displayed on screen. Now users know how to add correct value to complete the full command of `'set'`. One tip of saving typing here is to get last history entry by pressing `<Up>`. At last, the interactive status is set to `'SetParam'`.
- `write`: Output MPS/LP/CBF format model, COPT binary format model, IIS model, FeasRelax model, LP/MIP solution, optimal basis, settings of modified COPT parameters to file under given relative/absolution path. This command detects file types. For an instance, `'write diet.sol'` outputs LP solution to file `'diet.sol'`. An error message will be shown to users if file type do not match. Supported file types are: `'.mps'`, `'.lp'`, `'.bin'`, `'cbf'`, `'.iis'`, `'.relax'`, `'.sol'`, `'.bas'`, `'.mst'` and `'.par'`.
- `writemps`: Output current optimization model to a file of type `'.mps'`. Note `'.mps'` is appended to the file name, if users do not add it.
- `writelp`: Output current optimization model to a file of type `'.lp'`. Note `'.lp'` is appended to the file name, if users do not add it.
- `writecbf`: Output problem to a file of type `'.cbf'`. Note `'.cbf'` is appended to the file name, if users do not add it.
- `writebin`: Output current optimization model to a file of type `'.bin'`. Note `'.bin'` is appended to the file name, if users do not add it.

- `writeiis`: Output computed IIS model to a file of type `'.iis'`. Note `'.iis'` is appended to the file name, if users do not add it.
- `writerelax`: Output feasibility relaxation problem to a file of type `'.relax'`. Note `'.relax'` is appended to the file name, if users do not add it.
- `writesol`: Output LP solution of problem to a file of type `'.sol'`. Note `'.sol'` is appended to the file name, if users do not add it.
- `writepoolsol`: For the current problem, save the solution with the specified number in the solution pool to the result file under the given relative or absolute path. If the file extension is not```'.sol'``` , the suffix `'.sol'` is automatically added. The calling command is `'writepoolsol idx pool_idx.sol'`, where `'idx'` is the specified solution number in solution pool. Use the command `'get PoolSols'` to get the current number of solution pool results.
- `writebasis`: Output optimal basis to a file of type `'.bas'`. Note `'.bas'` is appended to the file name, if users do not add it.
- `writemst`: Output current best MIP solution to a file of type `'.mst'`. Note `'.mst'` is appended to the file name, if users do not add it.
- `writeparam`: Output modified COPT parameters to a file of type `'.par'`. Note `'.par'` is appended to the file name, if users do not add it.
- `writetuneparam`: Save the specified tuning result to the parameter setting file under the given relative or absolute path. The calling command is `'writetuneparam idx tune_idx.par'`, where `'idx'` is the specified tuning result number. Use the command `'get TuneResults'` to get the number of tuning results obtained after the current tuning.

3.5 Example Usage

This section shows how to use COPT command-line to interactively solve a well-known problem, called "Diet Problem". Please refer to *AMPL Interface - Example Usage* for problem description in more detail.

With valid license, COPT command-line should run as follows, after entering `copt_cmd` in command prompt.

```
copt_cmd
```

If users are new to COPT command-line, always start with shell command `'help'`.

```
Cardinal Optimizer v6.5.2. Build date Mar 22 2023
Copyright Cardinal Operations 2023. All Rights Reserved

COPT>
```

Suppose diet model has mps format and exits in the current working directory. In this way, we just type its file name to read, without worrying about its path.

```
COPT> read diet.mps
Reading from '/home/username/copt/diet.mps'
Reading finished (0.00s)
```

Before solving it, users are free to tune any COPT parameter. Below is an example to set value of double parameter `TimeLimit` to 10s.

If users are not familiar with COPT parameters, just type `'set'` to list all public COPT paramters and attributes with short description. Furthermore, `'set'` with parameter name, for example `'set TimeLimit'`, displays its current value, default value, min value and max value of the given parameter.

```
COPT> set timelimit 10

Setting parameter 'TimeLimit' to 10
```

After tuning parameters, it is time to solve the model. The messages during problem solving are shown as follows.

```
COPT> opt
Model fingerprint: 129c032d

Hardware has 4 cores and 8 threads. Using instruction set X86_NATIVE (1)
Minimizing an LP problem

The original problem has:
    4 rows, 8 columns and 31 non-zero elements
The presolved problem has:
    4 rows, 8 columns and 31 non-zero elements

Starting the simplex solver using up to 8 threads

Method   Iteration           Objective  Primal.NInf   Dual.NInf        Time
Dual             0    0.0000000000e+00            4           0       0.01s
Dual             1    8.8201991646e+01            0           0       0.01s
Postsolving
Dual             1    8.8200000000e+01            0           0       0.01s

Solving finished
Status: Optimal  Objective: 8.8200000000e+01  Iterations: 1  Time: 0.01s
```

After solving the model, users might check results by using shell command `'get'` with a parameter name. Note that, similar to `'set'`, type `'get'` to list all public parameters and attributes. In particular, `'get all'` shows all parameters/attributes and their current value.

```
COPT> get TimeLimit
  DblParam: [Current]  10s
COPT> get LpObjval
  DblAttr:  [Current] 88.2
COPT> get LpStatus
  IntAttr:  [Current] 1 optimal
```

Before leaving COPT command-line, users might output the model in format of mps, optimal basis, modified parameters, or LP solution to files. Below is an example to write LP solution to current directory.

```
COPT> writesol diet
  Writing solutions to /home/username/copt/diet.sol
COPT> quit
  Leaving COPT...
```

Below is the script file `diet.in` by putting everything together, see Listing 3.1:

Listing 3.1: `diet.in`

```
1 #COPT script-in file
2
3 read diet.mps
4 set timelimit 10
5 opt
6 writesol diet
7 quit
```

which is loaded by using the option `'-i'` when starting `copt_cmd`:

```
copt_cmd -i diet.in
```

or executing shell command `load` on fly.

```
COPT> load diet.in
```

Chapter 4

COPT Floating Licensing service

The **Cardinal Optimizer** provides COPT Floating Token Server on all supported platforms, who serve license tokens to COPT client applications over local network.

Once you have floating license properly installed, server owns a set of license tokens up to number described in the license file. Any properly configured COPT client of the same version can request a token from server and release it whenever quit.

4.1 Server Setup

The application of COPT Floating Token server includes `copt_flserver` executable and a configuration file `fls.ini`. The very first thing to do when server starts is to verify floating license locally, whose location is specified in `fls.ini`. If local validation passes, server connects to remote COPT licensing server for further validation, including machine IP, which is supposed to match IP range that user provided during registration. This means the machine running COPT Floating Token Server should have internet access in specified area. For details, please see descriptions below or refer to *How to obtain and setup license*.

4.1.1 Installation

The **Cardinal Optimizer** provides a separate package for remote services, which include COPT floating token server. Users may apply for remote package from customer service. Afterwards, unzip the remote package and move to any folder on your computer. The software is portable and does not change anything in the system it runs on. Below are details of installation.

Windows

Please unzip the remote package and move to any folder. Though, it is common to move to folder under `C:\Program Files`.

Linux

To unzip the remote package, enter the following command in terminal:

```
tar -xzf CardinalOptimizer-Remote-6.5.2-lnx64.tar.gz
```

Then, the following command moves folder copt_remote65 in current directory to other path. For an example, admin user may move it to folder under `/opt` and standard user may move it to `$HOME`.

```
sudo mv copt_remote65 /opt
```

Note that it requires `root` privilege to execute this command.

MacOS

To unzip the remote package, enter the following command in terminal:

```
tar -xzf CardinalOptimizer-Remote-6.5.2-osx64.tar.gz
```

Then, the following command moves folder copt_remote65 in current directory to other path. For an example, admin user may move it to folder under `/Applications` and standard user may move it to `$HOME`.

```
mv copt_remote65 /Applications
```

4.1.2 Floating License

After installing COPT remote package, it requirs floating license to run. It is prefered to save floating license files, `license.dat` and `license.key`, to `floating` folder in path of remote package.

Windows

If the COPT remote package is installed under `"C:\Program Files"`, execute the following command to enter `floating` folder in path of remote package.

```
cd "C:\Program Files\copt_remote65\floating"
```

Note that the tool `copt_licgen` creating license files exists under `tools` folder in path of remote package. The following command creates floating license files in current directory, given a floating license key, such as `7483dff0863ffdae9fff697d3573e8bc` .

```
..\tools\copt_licgen -key 7483dff0863ffdae9fff697d3573e8bc
```

Linux and MacOS

If the COPT remote package is installed under `"/Applications"`, execute the following command to enter `floating` folder in path of remote package on MacOS system.

```
cd /Applications/copt_remote65/floating
```

The following command creates floating license files in current directory, given a floating license key, such as `7483dff0863ffdae9fff697d3573e8bc` .

```
../tools/copt_licgen -key 7483dff0863ffdae9fff697d3573e8bc
```

In addition, if users run the above command when current directory is different than `floating` folder in path of remote package, it is prefered to move them to `floating`. The following command does so.

```
mv license.* /Application/copt_remote65/floating
```

4.1.3 Configuration

Below is a typical configuration file, `fls.ini`, of COPT Floating Token Server.

```
[Main]
Port=7979

[Licensing]
# if empty or default license name, it is from binary folder
# to get license files from cwd, add prefix "./"
# full path is supported as well
LicenseFile = license.dat
PubkeyFile = license.key

[Filter]
# default policy 0 indicates accepting all connections, except for ones in blacklist
# otherwise, denying all connections except for ones in whitelist
DefaultPolicy = 0
UseBlackList = true
```

(continues on next page)

(continued from previous page)

```
UseWhiteList = true
FilterListFile = flsfilters.ini
```

The `Main` section specifies port number, through which COPT clients connect to server and then obtain the license token. The `Licensing` section specifies location of floating license. As described in comments above, if emtpy string or default license file name is specified, floating license files are read from the binary folder where the server executable reside.

It is possible to run COPT Floating Token server, even if floating license files do not exist in the same folder as the server executive. One solution is to set `LicenseFile = ./license.dat` and `PubkeyFile = ./license.key`. By doing so, server read floating license from the current working directory. That is, user could execute server application at the path where floating license files exist.

The other solution is to set full path of license files in configuration. As mentioned before, Cardinal Optimizer allows users to set environment variable `COPT_LICENSE_DIR` for license files. For details, please refer to *How to install Cardinal Optimizer*. If user prefers the way of environment variable, the configuration file should have the full path to floating license.

In the `Filter` section, `DefaultPolicy` has default value 0, meaning all connections are accepted except for those in black lists; if it is set to non-zero value, then all connection are blocked except for those in white lists. In addition, black list is enabled if `UseBlackList` is true and white list is enabled if `UseWhiteList` is true. The filter configuration file is specified by `FilterListFile`. Below is an example of the filter configuration file.

```
[BlackList]
# 127.0.*.* + user@machine*

[WhiteList]
# 127.0.1.2/16 - user@machine*

[ToolList]
# only tool client at server side can access by default
127.0.0.1/32
```

It has three sections and each section has its own rules. In section of `BlackList`, one may add rules to block others from connection. In section of `WhiteList`, one may add rules to grant others for connection, even if the default policy is to block all connections. Only users listed in section of `ToolList` are able to connect to floating token server by Floating Token Server Managing Tool (see below for details).

Specifically, rules in filter configuration have format of starting with IP address. To specify IP range, you may include wildcard (*) in IP address, or use CIDR notation, that is, a IPv4 address and its associated network prefix. In addtion, a rule may include (+) or exclude (-) given user at given machine, such as `127.0.1.2/16 - user@machine`. Here, `user` refers to `username`, which can be queried by `whoami` on Linux/MacOS platform; `machine` refers to `computer name`, which can be queried by `hostname` on Linux/MacOS platform.

4.1.4 Example Usage

Suppose that floating license exists in the same folder where the server executable reside. To start the COPT Floating Token Server, just execute the following command at any directory in Windows console, or Linux/Mac terminal.

```
copt_flserver
```

If you see log information as follows, the Floating Token Server has been successfully started. Server monitors any connction from COPT clients, manages approved clients as well as requests in queue. User can stop Floating Token Server anytime when entering `q` or `Q`.

```
> copt_flserver
  [ Info] Floating Token Server, COPT v6.5.2 20230322
```

```
  [ Info] server started at port 7979
```

If failed to verify local floating license, or something is wrong on remote COPT license server, you might see error logs as follows.

```
> copt_flserver
  [ Info] Floating Token Server, COPT v6.5.2 20230322
  [Error] Invalid signature in public key file
  [Error] Fail to verify local license
```

and

```
> copt_flserver
  [ Info] Floating Token Server, COPT v6.5.2 20230322
  [Error] Error to connect license server
  [Error] Fail to verify floating license by server
```

4.2 Client Setup

COPT Clients can be COPT command-line tool, or any application which solve problems using COPT api, such as COPT python interface. Floating licensing is a better approach in terms of flexibility and efficiency. Different from stand-alone license, any machine having properly configured COPT client can legally run Cardingal Optimizer, as long as peak number of connected clients does not exceed the token number.

4.2.1 Configuration

Before running COPT as floating client, please make sure that you have installed COPT locally. For details, please refer to *How to install Cardinal Optimizer*. Users can skip obtaining local licenses by adding a floating configuration file `client.ini`.

Below is a typical configuration file, `client.ini`, of COPT floating clients.

```
Host = 192.168.1.11
Port = 7979
QueueTime = 600
```

As configured above, COPT floating client tries to connect to `192.168.1.11` at port `7979` with wait time in queue up to 600 seconds. Here, `Host` is set to `localhost` if empty or not specified; `QueueTime` is set to 0 if empty or not specified. Specifically, empty QueueTime means client does not wait and should quit immediately, if COPT Floating Token Server have no tokens available. Port number must be great than zero and should be the same as that specified in server configuration file. Note that keywords in the client configuration file are case insensitive.

Without local license files, a COPT application still works if client configuration file, `client.ini`, exists in one of the following three locations in order, that is, current working directory, environment directory by `COPT_LICENSE_DIR` and binary directory where COPT executable is located.

By design, COPT application reads local license files instead of client configuration file, if they both exist in the same location. On the other hand, if local license files are under the environment directory, to activate approach of floating licensing, user can simply add a configuration file, `client.ini`, under the current working directory (different from the environment directory).

If a COPT application calls COPT api to solve problems, such as COPT python interface, license is checked as soon as COPT environment object is created. If there exists proper client configuration file, `client.ini`, a license token is granted to COPT client. This license token is released and sent back to token server, as soon as last COPT environment object in the same process destroys.

4.2.2 Example Usage

Suppose that we've set client configuration file `client.ini` properly and have no local license, below is an example of obtaining a floating token by COPT command-line tool `copt_cmd`. Execute the following command in Windows console, or Linux/Mac terminal.

```
copt_cmd
```

If you see log information as follows, the COPT client, `copt_cmd`, has obtained the floating token successfully. COPT command-line tool is ready to solve optimization problems.

```
> copt_cmd
  Cardinal Optimizer v6.5.2. Build date Mar 22 2023
  Copyright Cardinal Operations 2023. All Rights Reserved

  [ Info] initialize floating client: ./client.ini

  [ Info] connecting to server ...
  [ Info] connection established
COPT>
```

If you see log information as follows, the COPT client, `copt_cmd`, has connected to COPT Floating Token Server. But due to limited number of tokens, client waits in queue of size 1.

```
> copt_cmd
  Cardinal Optimizer v6.5.2. Build date Mar 22 2023
  Copyright Cardinal Operations 2023. All Rights Reserved

  [ Info] Initialize floating client: ./client.ini

  [ Info] connecting to server ...
  [Error] empty license and queue size 1
  [ Info] Wait for license in  2 / 39 secs
  [ Info] Wait for license in  4 / 39 secs
  [ Info] Wait for license in  6 / 39 secs
  [ Info] Wait for license in  8 / 39 secs
  [ Info] Wait for license in 10 / 39 secs
  [ Info] Wait for license in 20 / 39 secs
  [ Info] Wait for license in 30 / 39 secs
```

If you see log information as follows, the COPT client, `copt_cmd`, has connected to COPT Floating Token Server. But client refused to wait in queue, as Queuetime is 0.

```
> copt_cmd
  Cardinal Optimizer v6.5.2. Build date Mar 22 2023
  Copyright Cardinal Operations 2023. All Rights Reserved

  [ Info] Initialize floating client: ./client.ini

  [ Info] connecting to server ...
  [Error] Server error: "no more token available", code = 2
  [Error] Fail to open: ./license.dat

  [Error] Fail to initialize cmdline
```

If you see log information as follows, the COPT client, `copt_cmd`, fails to connect to COPT Floating Token Server. Finally, client quits after time limit.

```
> copt_cmd
  Cardinal Optimizer v6.5.2. Build date Mar 22 2023
  Copyright Cardinal Operations 2023. All Rights Reserved
```

```
[ Info] initialize floating client: ./client.ini

[ Info] connecting to server ...
[ Info] wait for license in  2 / 10 secs
[ Info] wait for license in  4 / 10 secs
[ Info] wait for license in  6 / 10 secs
[ Info] wait for license in  8 / 10 secs
[ Info] wait for license in 10 / 10 secs
[Error] timeout at waiting for license
[Error] fail to open: ./license.dat

[Error] Fail to initialize cmdline
```

4.3 Floating Token Server Managing Tool

COPT floating token service ships with a tool `copt_flstool`, for retrieving information and tune parameters of floating token server on fly.

4.3.1 Tool Usage

Execute the following command in Windows console, Linux or MacOS terminal:

```
> copt_flstool
```

Below displays help messages of this tool:

```
> copt_flstool
  COPT Floating Token Server Managing Tool

  copt_flstool [-s server ip] [-p port] [-x passwd] command <param>

  commands are:   addblackrule <127.0.0.1/20[-user@machine]>
                  addwhiterule <127.0.*.*[+user@machine]>
                  getfilters
                  getinfo
                  resetfilters
                  setpasswd <xxx>
                  toggleblackrule <n-th>
                  togglewhiterule <n-th>
                  writefilters
```

If the `-s` and `-p` option are present, tool connects to floating token server with given server IP and port. Otherwise, tool connections to localhost and default port 7878. If floating token server sets a password, tool must provide password string after the `-x` option.

This tool provides the following commands:

- `AddBlackRule`: Add a new rule in black filters. each rule has format starting with non-empty IP address, which may have wildcard to match IPs in the scope. In addition, it is optional to be followed by including (+) or excluding (-) user name at machine name.
- `AddWhiteRule`: Add a new rule in white filters. Note that a white rule has the same format as a black rule.
- `GetFilters`: Get all rules of black filters, white filters and tool filters, along with relative sequence numbers, which are parameters for command ToggleBlackRule and ToggleWhiteRule.
- `GetInfo`: Get general information of floating token server, including token usage, connected clients, and all COPT versions in support.

- `ResetFilters`: Reset filter lists in memory to those on filter config file.

  – `SetPasswd`:

  Update password of target floating token server on fly.

  – `ToggleBlackRule`:

  Toggle between enabling and disabling a black rule, given its sequence number by GetFilters.

- `ToggleWhiteRule`: Toggle between enabling and disabling a white rule, given its sequence number by GetFilters.
- `WriteFilters`: Write filter lists in memory to filter config file.

4.3.2 Example Usage

The following command lists general information on local machine.

```
> copt_flstool GetInfo

[ Info] COPT Floating Token Server Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to localhost:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] general info
  # of available tokens is 3 / 3, queue size is 0
  # of active clients is 0
```

To run managing tool on other machine, its IP should be added to a rule in `ToolList` section in filter configuration file `flsfilters.ini`. The following command from other machine lists information of server 192.168.1.11.

```
> copt_flstool -s 192.168.1.11 GetInfo

[ Info] COPT Floating Token Server Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] general info
  # of available tokens is 3 / 3, queue size is 0
  # of active clients is 0
```

The following command shows all filter lists of server 192.168.1.11, including those in BlackList section, WhiteList section and ToolList section.

```
> copt_flstool -s 192.168.1.11 GetFilters

[ Info] COPT Floating Token Server Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] filters info
[BlackList]

[WhiteList]

[ToolList]
  [1]  127.0.0.1
```

The following command added user of IP 192.168.3.13 to black list.

```
> copt_flstool -s 192.168.1.11 AddBlackRule 192.168.3.133

[ Info] COPT Floating Token Server Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to floating token server
```

```
[ Info] [floating] server added new black rule (succeeded)
```

The follwing command shows that a new rule in BlackList section is added.

```
> copt_flstool -s 192.168.1.11 GetFilters

[ Info] COPT Floating Token Server Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] filters info
[BlackList]
  [1] 192.168.3.133

[WhiteList]

[ToolList]
  [1]  127.0.0.1
```

The following command disable a rule in BlackList section.

```
> copt_flstool -s 192.168.1.11 ToggleBlackRule 1

[ Info] COPT Floating Token Server Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to floating token server
[ Info] [floating] server toggle black rule [1] (succeeded)
```

4.4 Running as service

To run COPT floating token server as a system service, follow steps described in `readme.txt` under `floating` folder, and set config file `copt_flserver.service` properly.

Below is `readme.txt`, which lists installing steps in both Linux and MacOS platforms.

```
[Linux] To run copt_flserver as a service with systemd

Add a systemd file
    copy copt_flserver.service to /lib/systemd/system/
    sudo systemctl daemon-reload

Enable new service
    sudo systemctl start copt_flserver.service
    or
    sudo systemctl enable copt_flserver.service

Restart service
    sudo systemctl restart copt_flserver.service

Stop service
    sudo systemctl stop copt_flserver.service
    or
    sudo systemctl disable copt_flserver.service

Verify service is running
    sudo systemctl status copt_flserver.service

[MacOS] To run copt_flserver as a service with launchctrl

Add a plist file
    copy copt_flserver.plist to /Library/LaunchAgents as current user
```

(continues on next page)

(continued from previous page)

```
    or
    copy copt_flserver.plist to /Library/LaunchDaemons with the key 'UserName'

Enable new service
    sudo launchctl load -w /Library/LaunchAgents/copt_flserver.plist
    or
    sudo launchctl load -w /Library/LaunchDaemons/copt_flserver.plist

Stop service
    sudo launchctl unload -w /Library/LaunchAgents/copt_flserver.plist
    or
    sudo launchctl unload -w /Library/LaunchDaemons/copt_flserver.plist

Verify service is running
    sudo launchctl list shanshu.copt.flserver
```

4.4.1 Linux

Below are steps in details of how to run COPT floating token server as a system service in Linux platform.

For instance, assume that COPT remote service is installed under `'/home/eleven'`. In your terminal, type the following command to enter the root directory of floating service.

```
cd /home/eleven/copt_remote65/floating
```

modify template of the service config file `copt_flserver.service` in text format:

```
[Unit]
Description=COPT Floating Token Server

[Service]
WorkingDirectory=/path/to/service
ExecStart=/path/to/service/copt_flserver
Restart=always
RestartSec=1

[Install]
WantedBy=multi-user.target
```

That is, update template path in keyword `WorkingDirectory` and `ExecStart` to actual path where the floating service exits.

```
[Unit]
Description=COPT Floating Token Server

[Service]
WorkingDirectory=/home/eleven/copt_remote65/floating
ExecStart=/home/eleven/copt_remote65/floating/copt_flserver
Restart=always
RestartSec=1

[Install]
WantedBy=multi-user.target
```

Afterwards, copy `copt_flserver.service` to system service folder `/lib/systemd/system/` (see below).

```
sudo cp copt_flserver.service /lib/systemd/system/
```

The following command may be needed if you add or update service config file. It is not needed if service unit has been loaded before.

```
sudo systemctl daemon-reload
```

The following command starts the new floating service.

```
sudo systemctl start copt_flserver.service
```

To verify the floating service is actually running, type the following command

```
sudo systemctl status copt_flserver.service
```

If you see logs similar to below, COPT floating server is running successfully as a system service.

```
copt_flserver.service - COPT Floating Token Server
Loaded: loaded (/lib/systemd/system/copt_flserver.service; enabled; vendor preset:␣
↪enabled)
Active: active (running) since Tue 2021-06-29 11:46:10 CST; 3s ago
Main PID: 3054 (copt_flserver)
    Tasks: 6 (limit: 4915)
CGroup: /system.slice/copt_flserver.service
        └─3054 /home/eleven/copt_remote65/floating/copt_flserver

eleven-ubuntu systemd[1]: Started COPT Floating Token Server.
eleven-ubuntu COPTCLS[3054]: LWS: 4.1.4-b2011a00, loglevel 1039
eleven-ubuntu COPTCLS[3054]: NET CLI SRV H1 H2 WS IPv6-absent
eleven-ubuntu COPTCLS[3054]: server started at port 7979
```

To stop the floating service, type the following command

```
sudo systemctl stop copt_flserver.service
```

4.4.2 MacOS

Below are steps in details of how to run COPT floating token server as a system service in MacOS platform.

For instance, assume that COPT remote service is installed under `'/Applications'`. In your terminal, type the following command to enter the root directory of floating service.

```
cd /Applications/copt_remote65/floating
```

modify template of the service config file `copt_flserver.plist` in xml format:

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>shanshu.copt.flserver</string>
        <key>Program</key>
        <string>/path/to/service/copt_flserver</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

That is, update template path in `Program` tag to actual path where the floating service exits.

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
```

```
        <key>Label</key>
        <string>shanshu.copt.flserver</string>
        <key>Program</key>
        <string>/Applications/copt_remote65/floating/copt_flserver</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

Afterwards, copy `copt_flserver.plist` to system service folder `/Library/LaunchAgents` (see below).

```
sudo cp copt_flserver.plist /Library/LaunchAgents
```

The following command starts the new floating service.

```
sudo launchctl load -w /Library/LaunchAgents/copt_flserver.plist
```

To verify the floating service is actually running, type the following command

```
sudo launchctl list shanshu.copt.flserver
```

If you see logs similar to below, COPT floating server is running successfully as a system service.

```
{
    "LimitLoadToSessionType" = "System";
    "Label" = "shanshu.copt.flserver";
    "OnDemand" = false;
    "LastExitStatus" = 0;
    "PID" = 16406;
    "Program" = "/Applications/copt_remote65/floating/copt_flserver";
};
```

To stop the floating service, type the following command

```
sudo launchctl unload -w /Library/LaunchAgents/copt_flserver.plist
```

If the floating service should be run by a specific user, add `UserName` tag to conifg file. Below adds a user `eleven`, who has priviledge to run the floating service.

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>shanshu.copt.flserver</string>
        <key>Program</key>
        <string>/Applications/copt_remote65/floating/copt_flserver</string>
        <key>UserName</key>
        <string>eleven</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

Then copy new `copt_flserver.plist` to system service folder `/Library/LaunchDaemons` (see below).

```
sudo cp copt_flserver.plist /Library/LaunchDaemons
```

The following command starts the new floating service.

```
sudo launchctl load -w /Library/LaunchDaemons/copt_flserver.plist
```

To stop the floating service, type the following command

```
sudo launchctl unload -w /Library/LaunchDaemons/copt_flserver.plist
```

Chapter 5

COPT Compute Cluster Service

The **Cardinal Optimizer** provides COPT compute cluster service on all supported platforms, which allows you to offload optimization computations from COPT client applications over local network.

Once COPT compute cluster server runs at local network, any COPT client application with matching COPT version can connect to server and offload optimization computations. That is, COPT compute cluster clients are allowed to do modelling locally, execute optimization jobs remotely, and then obtain results interactively.

Note that the more computing power server has, the more optimization jobs can run simultaneously. Furthermore, COPT compute cluster service has functionality to cluster multiple servers together and therefore serve more COPT compute cluster clients over local network.

5.1 Server Setup

The COPT compute cluster service includes `copt_cluster` executable and a configuration file `cls.ini`. The very first thing to do when cluster server starts is to verify cluster license locally, whose path is specified in `cls.ini`. If local validation passes, cluster server might connect remotely to COPT licensing server for further validation, such as verifying machine IP, which is supposed to match IP range that user provided during registration. This means the server running COPT compute cluster service should have internet access in specified area, such as campus network. For details, please see descriptions below or refer to *How to obtain and setup license*.

5.1.1 Installation

The **Cardinal Optimizer** provides a separate package for remote services, which include COPT compute cluster. Users may apply for remote package from customer service. Afterwards, unzip the remote package and move to any folder on your computer. The software is portable and does not change anything in the system it runs on. Below are details of installation.

Windows

Please unzip the remote package and move to any folder. Though, it is common to move to folder under `C:\Program Files`.

Linux

To unzip the remote package, enter the following command in terminal:

```
tar -xzf CardinalOptimizer-Remote-6.5.2-lnx64.tar.gz
```

Then, the following command moves folder copt_remote65 in current directory to other path. For an example, admin user may move it to folder under `/opt` and standard user may move it to `$HOME`.

```
sudo mv copt_remote65 /opt
```

Note that it requires `root` privilege to execute this command.

MacOS

To unzip the remote package, enter the following command in terminal:

```
tar -xzf CardinalOptimizer-Remote-6.5.2-osx64.tar.gz
```

Then, the following command moves folder copt_remote65 in current directory to other path. For an example, admin user may move it to folder under `/Applications` and standard user may move it to `$HOME`.

```
mv copt_remote65 /Applications
```

If you see errors below or similar signature problem of COPT lib during installation,

```
"libcopt.dylib" cannot be opened because the developer cannot be verified.
macOS cannot verify that this app is free from malware.
```

run the following command as root user, to bypass check of loading dynamic lib on MacOS.

```
xattr -d com.apple.quarantine CardinalOptimizer-Remote-6.5.2-osx64.tar.gz
```

or

```
xattr -dr com.apple.quarantine /Applications/copt_remote65
```

5.1.2 Cluster License

After installing COPT remote package, it requirs cluster license to run. It is prefered to save cluster license files, `license.dat` and `license.key`, to `cluster` folder in path of remote package.

Windows

If the COPT remote package is installed under `"C:\Program Files"`, execute the following command to enter `cluster` folder in path of remote package.

```
cd "C:\Program Files\copt_remote65\cluster"
```

Note that the tool `copt_licgen` creating license files exists under `tools` folder in path of remote package. The following command creates cluster license files in current directory, given a cluster license key, such as `7483dff0863ffdae9fff697d3573e8bc` .

```
..\tools\copt_licgen -key 7483dff0863ffdae9fff697d3573e8bc
```

Linux and MacOS

If the COPT remote package is installed under `"/Applications"`, execute the following command to enter `cluster` folder in path of remote package on MacOS system.

```
cd /Applications/copt_remote65/cluster
```

The following command creates cluster license files in current directory, given a cluster license key, such as `7483dff0863ffdae9fff697d3573e8bc` .

```
../tools/copt_licgen -key 7483dff0863ffdae9fff697d3573e8bc
```

In addition, if users run the above command when current directory is different than `cluster` folder in path of remote package, it is prefered to move them to `cluster`. The following command does so.

```
mv license.* /Application/copt_remote65/cluster
```

5.1.3 Configuration

Below is a typical configuration file, `cls.ini`, of COPT compute cluster.

```
[Main]
Port = 7878
# Number of total tokens, what copt jobs can run simutaneously up to.
NumToken = 3
# Password is case-sensitive and default is emtpy;
# It applies to both copt clients and cluster nodes.
PassWd =
# Data folder of cluster relative to its binary folder,
# where multiple versions of copt libraries and related licenses reside.
DataFolder = ./data

[SSL]
# Needed if connecting using SSL/TLS
CaFile =
CertFile =
CertkeyFile =

[Licensing]
# If empty or default license name, it is from binary folder;
# To get license files from cwd, add prefix "./";
# Full path is supported as well.
LicenseFile = license.dat
PubkeyFile = license.key

[Cluster]
# Host name and port of parent node in cluster.
# Default is empty, meaning not connecting to other node.
Parent =
PPort = 7878

[Filter]
# default policy 0 indicates accepting all connections, except for ones in blacklist
# otherwise, denying all connections except for ones in whitelist
DefaultPolicy = 0
UseBlackList = true
UseWhiteList = true
FilterListFile = clsfilters.ini
```

The `Main` section specifies port number, through which COPT compute cluster clients connect to server; token number, the number of optimization jobs that server can run simultaneously up to; password string, if specified, cluster clients should send the same password when requesting for service.

The COPT compute cluster may install multiple versions of COPT to subfolder of `DataFolder`. Only clients with matching version (major and minor) will get approved and then offload optimization jobs at server side. Note that the COPT compute cluster pre-installs a COPT solver of the same version as server itself, which illustrate how to install other versions of COPT.

For instance, the COPT compute cluster has default COPT v6.5.2 installed and users plan to install COPT of other version v4.0.7. users may create a folder ./data/copt/4.0.7/ and copy a COPT C lib of the same version to it. Specifically, on Linux platform, copy C dynamic library `libcopt.so` from the binary folder `$COPT_HOME/lib/` of COPT v4.0.7 to subfolder ./data/copt/4.0.7/ of the COPT compute cluster.

Furthermore, users are allowed to install newer version of COPT than cluster server version, such as COPT v7.0.0. To do so, follow the same step of copying a C lib of COPT v7.0.0 to ./data/copt/7.0.0/. In addition, users need a personal license of v7.0.0 to load C lib of COPT v7.0.0 at server side. That is, copy valid personal license files to folder ./data/copt/7.0.0/ as well. However, this simple procedure may break if the newer COPT solver has significant updates. In this case, it is necessary to upgrade the COPT compute cluster to newer version, that is, v7.0.0.

Below is an example of directory structure of cluster server on Linux platform. It includes pre-installed COPT v6.5.2, COPT of previous version v4.0.7, and COPT of newer version v7.0.0.

```
~/copt_remote65/cluster
│   cls.ini
│   copt_cluster
│   license.dat -> cluster license v6.5.2
│   license.key
│
└─data
    └─copt
        └─4.0.7
        │   libcopt.so
        └─6.5.2
        │   libcopt.so
        └─7.0.0
            libcopt.so
            license.dat -> license v7.0.0
            license.key
```

The `Licensing` section specifies location of cluster license files. As described in comments above, if emtpy string or default license file name is specified, cluster license files are read from the binary folder where the cluster executable reside.

It is possible to run COPT compute cluster service, even if cluster license files do not exist in the same folder as the cluster executive. One option is to set `LicenseFile = ./license.dat` and `PubkeyFile = ./license.key`. By doing so, the COPT compute cluster reads cluster license files from the current working directory. That is, user could execute command at the path where cluster license files exist to run service.

The other option is to set full path of license files in configuration. As mentioned before, Cardinal Optimizer allows user to set environment variable `COPT_LICENSE_DIR` for license files. For details, please refer to *How to install Cardinal Optimizer*. If user prefers the way of environment variable, `cls.ini` should have the full path to cluster license files.

The `Cluster` section specifies IP and port number of parent node. By connecting to parent node, this server joins a cluster of servers running COPT compute cluster service. The default value is empty, which means this server does not join other cluster group.

In the `Filter` section, `DefaultPolicy` has default value 0, meaning all connections are accepted except for those in black lists; if it is set to non-zero value, then all connection are blocked except for those in white lists. In addition, black list is enabled if `UseBlackList` is true and white list is enabled if `UseWhiteList` is true. The filter configuration file is specified by `FilterListFile`. Below is an example of the filter configuration file.

```
[BlackList]
# 127.0.*.* + user@machine*

[WhiteList]
# 127.0.1.2/16 - user@machine*

[ToolList]
# only tool client at server side can access by default
127.0.0.1/32
```

It has three sections and each section has its own rules. In section of `BlackList`, one may add rules to block others from connection. In section of `WhiteList`, one may add rules to grant others for connection, even if the default policy is to block all connections. Only users listed in section of `ToolList` are able to connect to cluster server by Cluster Managing Tool (see below for details).

Specifically, rules in filter configuration have format of starting with IP address. To specify IP range, you may include wildcard (*) in IP address, or use CIDR notation, that is, a IPv4 address and its associated network prefix. In addtion, a rule may include (+) or exclude (-) given user at given machine, such

as `127.0.1.2/16 - user@machine`. Here, `user` refers to `username`, which can be queried by `whoami` on Linux/MacOS platform; `machine` refers to `computer name`, which can be queried by `hostname` on Linux/MacOS platform.

5.1.4 Example Usage

Suppose that cluster license files exist in the same folder where the cluster executable reside. To start the COPT compute cluster, just execute the following command at any directory in Windows console, or Linux/Mac terminal.

```
copt_cluster
```

If you see log information as follows, the COPT compute cluster has been successfully started. Server monitors any connection from COPT compute cluster clients, manages client requests in queue as well as approved clients. User may stop cluster server anytime when entering `q` or `Q`.

```
> copt_cluster
  [ Info] start COPT Compute Cluster, COPT v6.5.2 20230322
  [ Info] [NODE] node has been initialized
  [ Info] server started at port 7878
```

If failed to verify local cluster license, or something is wrong on remote COPT licensing server, you might see error logs as follows.

```
> copt_cluster
  [ Info] start COPT Compute Cluster, COPT v6.5.2 20230322
  [Error] Invalid signature in public key file
  [Error] Fail to verify local license
```

and

```
> copt_cluster
  [ Info] start COPT Compute Cluster, COPT v6.5.2 20230322
  [Error] Error to connect license server
  [Error] Fail to verify cluster license by server
```

5.2 Client Setup

The COPT compute cluster client can be COPT command-line, or any application which solves problems by COPT API, such as COPT cpp/java/csharp/python interface. The COPT compute cluster service is a better approach in terms of flexibility and efficiency. Any COPT compute cluster client can legally run Cardingal Optimizer without local license.

5.2.1 Configuration

Before running COPT as cluster client, please make sure that you have installed COPT locally. For details, please refer to *How to install Cardinal Optimizer*. Users can skip obtaining local licenses by adding a cluster configuration file `client.ini`.

Below is a typical configuration file, `client.ini`, of COPT compute cluster client.

```
Cluster = 192.168.1.11
Port = 7878
QueueTime = 600
Passwd =
```

As configured above, COPT compute cluster client tries to connect to `192.168.1.11` at port `7878` with waiting time in queue up to 600 seconds. Here, the default value of `Cluster` is localhost; `QueueTime` or

`WaitTime` is set to 0 if empty or not specified. Specifically, empty `QueueTime` means client does not wait and should quit immediately, if the COPT compute cluster have no more token available. `Port` number must be great than zero if specified and should be the same as that specified in cluster configuration file `cls.ini`. Note that keywords in the configuration file are case insensitive.

To run as a COPT compute cluster client, an application must have configuration file, `client.ini`, in one of the following three locations, that is, current working directory, environment directory by `COPT_LICENSE_DIR` and binary directory where COPT executable resides.

By design, COPT application reads local license files instead of `client.ini`, if they both exist in the same location. However, if local license files are under the environment directory, to connect to cluster server, user could simply add a configuration file, `client.ini`, under the current working directory (different from the environment directory).

If a COPT application calls COPT API to solve problems, such as COPT python interface, license is checked as soon as COPT environment object is created. If there only exists proper configuration file, `client.ini`, the application works as a COPT compute cluster client and obtains token to offload optimization jobs. As soon as COPT environment object is destroyed, the COPT compute cluster server is notified to release token and thus to approve more requests waiting in queue.

5.2.2 Example Usage

Suppose that we've set configuration file `client.ini` properly and have no local license, below is an example of connecting to cluster server by COPT command-line tool `copt_cmd`. Execute the following command in Windows console, or Linux/Mac terminal.

```
copt_cmd
```

If you see log information as follows, the COPT compute cluster client, `copt_cmd`, has connected to cluster server successfully. COPT command-line tool is ready to do modelling locally and then offload optimization jobs to server.

```
> copt_cmd
  Cardinal Optimizer v6.5.2. Build date Mar 22 2023
  Copyright Cardinal Operations 2023. All Rights Reserved

  [ Info] initialize cluster client with ./client.ini

  [ Info] wait for server in 0 / 39 secs
  [ Info] connecting to cluster server 192.168.1.11:7878
COPT>
```

If you see log information as follows, the COPT compute cluster client, `copt_cmd`, has connected to cluster server. However, due to limited number of tokens, it waits in queue of size 5, until timeout.

```
> copt_cmd
  Cardinal Optimizer v6.5.2. Build date Mar 22 2023
  Copyright Cardinal Operations 2023. All Rights Reserved

  [ Info] initialize cluster client with ./client.ini

  [ Info] wait for server in 0 / 39 secs
  [ Info] connecting to cluster server 192.168.1.11:7878

  [ Warn] wait in queue of size 5
  [ Info] wait for license in  2 / 39 secs
  [ Info] wait for license in  4 / 39 secs
  [ Info] wait for license in  6 / 39 secs
  [ Info] wait for license in  8 / 39 secs
  [ Info] wait for license in 10 / 39 secs
  [ Info] wait for license in 20 / 39 secs
```

```
  [ Info] wait for license in 30 / 39 secs
  [Error] timeout at waiting for server approval
  [Error] Fail to initialize copt command-line tool
```

If you see log information as follows, the COPT compute cluster client, `copt_cmd`, has connected to cluster server. But it refused to wait in queue, as Queuetime was set to 0. Therefore, client quits with error immediately.

```
> copt_cmd
  Cardinal Optimizer v6.5.2. Build date Mar 22 2023
  Copyright Cardinal Operations 2023. All Rights Reserved

  [ Info] initialize cluster client with ./client.ini

  [ Info] wait for server in 0 / 9 secs
  [ Info] connecting to cluster server 192.168.1.11:7878
  [ Warn] server error: "no more token available", code = 129
  [Error] Fail to initialize copt command-line tool
```

If you see log information as follows, the COPT compute cluster client, `copt_cmd`, fails to connect to cluster server. Finally, client quits after timeout.

```
> copt_cmd
  Cardinal Optimizer v6.5.2. Build date Mar 22 2023
  Copyright Cardinal Operations 2023. All Rights Reserved

  [ Info] initialize cluster client with ./client.ini

  [ Info] wait for server in 0 / 39 secs
  [ Info] connecting to cluster server 192.168.1.11:7878
  [ Info] wait for license in  2 / 39 secs
  [ Info] wait for license in  4 / 39 secs
  [ Info] wait for license in  6 / 39 secs
  [ Info] wait for license in  8 / 39 secs
  [ Info] wait for license in 10 / 39 secs
  [ Info] wait for license in 20 / 39 secs
  [ Info] wait for license in 30 / 39 secs
  [Error] timeout at waiting for server approval
  [Error] Fail to initialize copt command-line tool
```

5.3 COPT Cluster Managing Tool

COPT cluster service ships with a tool `copt_clstool`, for retrieving information and tune parameters of cluster servers on fly.

5.3.1 Tool Usage

Execute the following command in Windows console, Linux or MacOS terminal:

```
> copt_clstool
```

Below displays help messages of this tool:

```
> copt_clstool
  COPT Cluster Managing Tool

  copt_clstool [-s server ip] [-p port] [-x passwd] command <param>
```

(continues on next page)

(continued from previous page)

```
commands are:   addblackrule <127.0.0.1/20[-user@machine]>
                addwhiterule <127.0.*.*[+user@machine]>
                getfilters
                getinfo
                getnodes
                reload
                resetfilters
                setparent <xxx:7878>
                setpasswd <xxx>
                settoken  <num>
                toggleblackrule <n-th>
                togglewhiterule <n-th>
                writefilters
```

If the `-s` and `-p` option are present, tool connects to cluster server with given server IP and port. Otherwise, tool connections to localhost and default port 7878. If cluster server sets a password, tool must provide password string after the `-x` option.

This tool provides the following commands:

- `AddBlackRule`: Add a new rule in black filters. each rule has format starting with non-empty IP address, which may have wildcard to match IPs in the scope. In addition, it is optional to be followed by including (+) or excluding (-) user name at machine name.
- `AddWhiteRule`: Add a new rule in white filters. Note that a white rule has the same format as a black rule.
- `GetFilters`: Get all rules of black filters, white filters and tool filters, along with relative sequence numbers, which are parameters for command ToggleBlackRule and ToggleWhiteRule.
- `GetInfo`: Get general information of cluster server, including token usage, connected clients, and all COPT versions in support.
- `GetNodes`: Get information of nodes in cluster, including parent address and status, all children nodes.
- `Reload`: Reload available token information of all child nodes, in case it is not consistent for various reasons.
- `ResetFilters`: Reset filter lists in memory to those on filter config file.
- `SetParent`: Change parent node address on fly and then connecting to new parent. In this way, it avoids `draining` operation when stopping a node for maintenance purpose.
- `SetPasswd`: Update password of target cluster server on fly.
- `SetToken`: Change token number of target cluster server on fly.
- `ToggleBlackRule`: Toggle between enabling and disabling a black rule, given its sequence number by GetFilters.
- `ToggleWhiteRule`: Toggle between enabling and disabling a white rule, given its sequence number by GetFilters.
- `WriteFilters`: Write filter lists in memory to filter config file.

5.3.2 Example Usage

The following command lists general information on local machine.

```
> copt_clstool GetInfo

[ Info] COPT Cluster Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to localhost:7878
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] general info
  # of available tokens is 3 / 3, queue size is 0
  # of active clients is 0
  # of installed COPT versions is 1
    COPT v6.5.2
```

To run managing tool on other machine, its IP should be added to a rule in `ToolList` section in filter configuration file `clsfilters.ini`. The following command from other machine lists cluster information of server 192.168.1.11.

```
> copt_clstool -s 192.168.1.11 GetNodes

[ Info] COPT Cluster Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7878
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] node info
  [Parent] (null):7878 (Lost)
  [Child] Node_192.168.1.12:7878_N0001, v2.0=3
  Total num of child nodes is 1
```

The following command changes token number of server 192.168.1.11 from 3 to 0.

```
> copt_clstool -s 192.168.1.11 SetToken 0

[ Info] COPT Cluster Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7878
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] total token was 3 and now set to 0
```

The following command shows all filter lists of server 192.168.1.11, including those in BlackList section, WhiteList section and ToolList section.

```
> copt_clstool -s 192.168.1.11 GetFilters

[ Info] COPT Cluster Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] filters info
[BlackList]

[WhiteList]

[ToolList]
  [1]  127.0.0.1
```

The following command added user of IP 192.168.3.13 to black list.

```
> copt_clstool -s 192.168.1.11 AddBlackRule 192.168.3.133

[ Info] COPT Cluster Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] server added new black rule (succeeded)
```

The follwing command shows that a new rule in BlackList section is added.

```
> copt_clstool -s 192.168.1.11 GetFilters

[ Info] COPT Cluster Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] filters info
[BlackList]
  [1] 192.168.3.133

[WhiteList]

[ToolList]
  [1]  127.0.0.1
```

The following command disable a rule in BlackList section.

```
> copt_clstool -s 192.168.1.11 ToggleBlackRule 1

[ Info] COPT Cluster Managing Tool, COPT v6.5.2 20230322
[ Info] connecting to 192.168.1.11:7979
[ Info] [command] wait for connecting to cluster
[ Info] [cluster] server toggle black rule [1] (succeeded)
```

5.4 Running as service

To run COPT compute cluster server as a system service, follow steps described in `readme.txt` under `cluster` folder, and set config file `copt_cluster.service` properly.

Below is `readme.txt`, which lists installing steps in both Linux and MacOS platforms.

```
[Linux] To run copt_cluster as a service with systemd

Add a systemd file
    copy copt_cluster.service to /lib/systemd/system/
    sudo systemctl daemon-reload

Enable new service
    sudo systemctl start copt_cluster.service
    or
    sudo systemctl enable copt_cluster.service

Restart service
    sudo systemctl restart copt_cluster.service

Stop service
    sudo systemctl stop copt_cluster.service
    or
    sudo systemctl disable copt_cluster.service

Verify service is running
    sudo systemctl status copt_cluster.service

[MacOS] To run copt_cluster as a service with launchctrl

Add a plist file
    copy copt_cluster.plist to /Library/LaunchAgents as current user
    or
    copy copt_cluster.plist to /Library/LaunchDaemons with the key 'UserName'
```

(continues on next page)

(continued from previous page)

```
Enable new service
    sudo launchctl load -w /Library/LaunchAgents/copt_cluster.plist
    or
    sudo launchctl load -w /Library/LaunchDaemons/copt_cluster.plist

Stop service
    sudo launchctl unload -w /Library/LaunchAgents/copt_cluster.plist
    or
    sudo launchctl unload -w /Library/LaunchDaemons/copt_cluster.plist

Verify service is running
    sudo launchctl list shanshu.copt.cluster
```

5.4.1 Linux

Below are steps in details of how to run COPT compute cluster server as a system service in Linux platform.

For instance, assume that COPT remote service is installed under `'/home/eleven'`. In your terminal, type the following command to enter the root directory of cluster service.

```
cd /home/eleven/copt_remote65/cluster
```

modify template of the service config file `copt_cluster.service` in text format:

```
[Unit]
Description=COPT Compute Cluster Server

[Service]
WorkingDirectory=/path/to/service
ExecStart=/path/to/service/copt_cluster
Restart=always
RestartSec=1

[Install]
WantedBy=multi-user.target
```

That is, update template path in keyword `WorkingDirectory` and `ExecStart` to actual path where the cluster service exits.

```
[Unit]
Description=COPT Compute Cluster Server

[Service]
WorkingDirectory=/home/eleven/copt_remote65/cluster
ExecStart=/home/eleven/copt_remote65/cluster/copt_cluster
Restart=always
RestartSec=1

[Install]
WantedBy=multi-user.target
```

Afterwards, copy `copt_cluster.service` to system service folder `/lib/systemd/system/` (see below).

```
sudo cp copt_cluster.service /lib/systemd/system/
```

The following command may be needed if you add or update service config file. It is not needed if service unit has been loaded before.

```
sudo systemctl daemon-reload
```

The following command starts the new cluster service.

```
sudo systemctl start copt_cluster.service
```

To verify the cluster service is actually running, type the following command

```
sudo systemctl status copt_cluster.service
```

If you see logs similar to below, COPT compute cluster server is running successfully as a system service.

```
copt_cluster.service - COPT Cluster Server
Loaded: loaded (/lib/systemd/system/copt_cluster.service; enabled; vendor preset:␣
↪enabled)
Active: active (running) since Sat 2021-08-28 11:46:10 CST; 3s ago
Main PID: 3054 (copt_cluster)
    Tasks: 6 (limit: 4915)
CGroup: /system.slice/copt_cluster.service
          └─3054 /home/eleven/copt_remote65/cluster/copt_cluster

eleven-ubuntu systemd[1]: Started COPT Cluster Server.
eleven-ubuntu COPTCLS[3054]: LWS: 4.1.4-b2011a00, loglevel 1039
eleven-ubuntu COPTCLS[3054]: NET CLI SRV H1 H2 WS IPv6-absent
eleven-ubuntu COPTCLS[3054]: server started at port 7878
eleven-ubuntu COPTCLS[3054]: LWS: 4.1.4-b2011a00, loglevel 1039
eleven-ubuntu COPTCLS[3054]: NET CLI SRV H1 H2 WS IPv6-absent
eleven-ubuntu COPTCLS[3054]: [NODE] node has been initialized
```

To stop the cluster service, type the following command

```
sudo systemctl stop copt_cluster.service
```

5.4.2 MacOS

Below are steps in details of how to run COPT Compute Cluster server as a system service in MacOS platform.

For instance, assume that COPT remote service is installed under `"/Applications"`. In your terminal, type the following command to enter the root directory of cluster service.

```
cd /Applications/copt_remote65/cluster
```

modify template of the service config file `copt_cluster.plist` in xml format:

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>shanshu.copt.cluster</string>
        <key>Program</key>
        <string>/path/to/service/copt_cluster</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

That is, update template path in `Program` tag to actual path where the cluster service exits.

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>shanshu.copt.cluster</string>
        <key>Program</key>
        <string>/Applications/copt_remote65/cluster/copt_cluster</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

Afterwards, copy `copt_cluster.plist` to system service folder `/Library/LaunchAgents` (see below).

```
sudo cp copt_cluster.plist /Library/LaunchAgents
```

The following command starts the new cluster service.

```
sudo launchctl load -w /Library/LaunchAgents/copt_cluster.plist
```

To verify the cluster service is actually running, type the following command

```
sudo launchctl list shanshu.copt.cluster
```

If you see logs similar to below, COPT compute cluster server is running successfully as a system service.

```
{
    "LimitLoadToSessionType" = "System";
    "Label" = "shanshu.copt.cluster";
    "OnDemand" = false;
    "LastExitStatus" = 0;
    "PID" = 16406;
    "Program" = "/Applications/copt_remote65/cluster/copt_cluster";
};
```

To stop the cluster service, type the following command

```
sudo launchctl unload -w /Library/LaunchAgents/copt_cluster.plist
```

If the cluster service should be run by a specific user, add `UserName` tag to conifg file. Below adds a user `eleven`, who has priviledge to run the cluster service.

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>shanshu.copt.cluster</string>
        <key>Program</key>
        <string>/Applications/copt_remote65/cluster/copt_cluster</string>
        <key>UserName</key>
        <string>eleven</string>
        <key>RunAtLoad</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
    </dict>
</plist>
```

Then copy new `copt_cluster.plist` to system service folder `/Library/LaunchDaemons` (see below).

```
sudo cp copt_cluster.plist /Library/LaunchDaemons
```

The following command starts the new cluster service.

```
sudo launchctl load -w /Library/LaunchDaemons/copt_cluster.plist
```

To stop the cluster service, type the following command

```
sudo launchctl unload -w /Library/LaunchDaemons/copt_cluster.plist
```

Chapter 6

COPT Quick Start

6.1 C Interface

This chapter illustrate the use of C interface of Cardinal Optimizer through a simple C example. The problem to solve is shown in Eq. 6.1:

$$
\begin{aligned}
&\text{Maximize:} \\
&\qquad 1.2x + 1.8y + 2.1z \\
&\text{Subject to:} \\
&\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\
&\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\
&\text{Bounds:} \\
&\qquad 0.1 \leq x \leq 0.6 \\
&\qquad 0.2 \leq y \leq 1.5 \\
&\qquad 0.3 \leq z \leq 2.8
\end{aligned}
\tag{6.1}
$$

6.1.1 Example details

The source code for solving the above problem using C API of Cardinal Optimizer is shown in Listing 6.1:

Listing 6.1: `lp_ex1.c`

```c
/*
 * This file is part of the Cardinal Optimizer, all rights reserved.
 */

// The problem to solve:
//
//  Maximize:
//    1.2 x + 1.8 y + 2.1 z
//
//  Subject to:
//    1.5 x + 1.2 y + 1.8 z <= 2.6
//    0.8 x + 0.6 y + 0.9 z >= 1.2
//
//  where:
//    0.1 <= x <= 0.6
//    0.2 <= y <= 1.5
//    0.3 <= z <= 2.8
```

(continues on next page)

(continued from previous page)

```c

#include "copt.h"

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int errcode = 0;

  copt_env *env = NULL;
  copt_prob *prob = NULL;

  // Create COPT environment
  errcode = COPT_CreateEnv(&env);
  if (errcode) goto COPT_EXIT;

  // Create COPT problem
  errcode = COPT_CreateProb(env, &prob);
  if (errcode) goto COPT_EXIT;

  // Add variables
  //
  //   obj: 1.2 C0 + 1.8 C1 + 2.1 C2
  //
  //   var:
  //     0.1 <= C0 <= 0.6
  //     0.2 <= C1 <= 1.5
  //     0.3 <= C2 <= 2.8
  //
  int ncol = 3;
  double colcost[] = {1.2, 1.8, 2.1};
  double collb[] = {0.1, 0.2, 0.3};
  double colub[] = {0.6, 1.5, 1.8};

  errcode = COPT_AddCols(prob, ncol, colcost, NULL, NULL, NULL, NULL,
                         NULL, collb, colub, NULL);
  if (errcode) goto COPT_EXIT;

  // Add constraints
  //
  //   r0: 1.5 C0 + 1.2 C1 + 1.8 C2 <= 2.6
  //   r1: 0.8 C0 + 0.6 C1 + 0.9 C2 >= 1.2
  //
  int nrow = 2;
  int rowbeg[] = {0, 3};
  int rowcnt[] = {3, 3};
  int rowind[] = {0, 1, 2, 0, 1, 2};
  double rowelem[] = {1.5, 1.2, 1.8, 0.8, 0.6, 0.9};
  char rowsen[] = {COPT_LESS_EQUAL, COPT_GREATER_EQUAL};
  double rowrhs[] = {2.6, 1.2};

  errcode = COPT_AddRows(prob, nrow, rowbeg, rowcnt, rowind, rowelem,
                         rowsen, rowrhs, NULL, NULL);
  if (errcode) goto COPT_EXIT;

  // Set parameters and attributes
  errcode = COPT_SetDblParam(prob, COPT_DBLPARAM_TIMELIMIT, 10);
  if (errcode) goto COPT_EXIT;
  errcode = COPT_SetObjSense(prob, COPT_MAXIMIZE);
  if (errcode) goto COPT_EXIT;

```

(continues on next page)

(continued from previous page)

```c
  // Solve problem
  errcode = COPT_SolveLp(prob);
  if (errcode) goto COPT_EXIT;

  // Analyze solution
  int lpstat = COPT_LPSTATUS_UNSTARTED;
  double lpobjval;
  double *lpsol = NULL;
  int *colstat = NULL;

  errcode = COPT_GetIntAttr(prob, COPT_INTATTR_LPSTATUS, &lpstat);
  if (errcode) goto COPT_EXIT;

  if (lpstat == COPT_LPSTATUS_OPTIMAL) {
    lpsol = (double *) malloc(ncol * sizeof(double));
    colstat = (int *) malloc(ncol * sizeof(int));

    errcode = COPT_GetLpSolution(prob, lpsol, NULL, NULL, NULL);
    if (errcode) goto COPT_EXIT;

    errcode = COPT_GetBasis(prob, colstat, NULL);
    if (errcode) goto COPT_EXIT;

    errcode = COPT_GetDblAttr(prob, COPT_DBLATTR_LPOBJVAL, &lpobjval);
    if (errcode) goto COPT_EXIT;

    printf("\nObjective value: %.6f\n", lpobjval);

    printf("Variable solution: \n");
    for (int i = 0; i < ncol; ++i)
      printf("  x[%d] = %.6f\n", i, lpsol[i]);

    printf("Variable basis status: \n");
    for (int i = 0; i < ncol; ++i)
      printf("  x[%d]: %d\n", i, colstat[i]);

    free(lpsol);
    free(colstat);
  }

  // Write problem, solution and modified parameters to files
  errcode = COPT_WriteMps(prob, "lp_ex1.mps");
  if (errcode) goto COPT_EXIT;
  errcode = COPT_WriteBasis(prob, "lp_ex1.bas");
  if (errcode) goto COPT_EXIT;
  errcode = COPT_WriteSol(prob, "lp_ex1.sol");
  if (errcode) goto COPT_EXIT;
  errcode = COPT_WriteParam(prob, "lp_ex1.par");
  if (errcode) goto COPT_EXIT;

  // Error handling
COPT_EXIT:
  if (errcode) {
    char errmsg[COPT_BUFFSIZE];

    COPT_GetRetcodeMsg(errcode, errmsg, COPT_BUFFSIZE);
    printf("ERROR %d: %s\n", errcode, errmsg);

    return 0;
  }

```

(continues on next page)

(continued from previous page)

```
  // Delete problem and environment
  COPT_DeleteProb(&prob);

  COPT_DeleteEnv(&env);

  return 0;
}
```

We will explain how to use the C API step by step based on code above, please refer to *C API Reference* for detailed usage of C API.

Creating the environment

To solve any problem with Cardinal Optimizer, users are required to create optimization environment first, and check if it was created successfully by checking the return value:

```
// Create COPT environment
errcode = COPT_CreateEnv(&env);
if (errcode) goto COPT_EXIT;
```

If non-zero value was returned, it will jump to error reporting code block for detailed information and exit.

Creating the problem

Once the optimization environment was successfully created, users will need to create problem then, the problem is the main structure that consists of variables, constraints etc. Users need to check the return value too.

```
// Create COPT problem
errcode = COPT_CreateProb(env, &prob);
if (errcode) goto COPT_EXIT;
```

If non-zero value was returned, it will jump to error reporting code block for detailed information and exit.

Adding variables

For linear problem, C API allows users to specify costs of variables in objective, and lower and upper bound simultaneously. For the problem above, we use code below to create variables:

```
// Add variables
//
//   obj: 1.2 C0 + 1.8 C1 + 2.1 C2
//
//   var:
//     0.1 <= C0 <= 0.6
//     0.2 <= C1 <= 1.5
//     0.3 <= C2 <= 2.8
//
int ncol = 3;
double colcost[] = {1.2, 1.8, 2.1};
double collb[] = {0.1, 0.2, 0.3};
double colub[] = {0.6, 1.5, 1.8};

errcode = COPT_AddCols(prob, ncol, colcost, NULL, NULL, NULL, NULL,
                       NULL, collb, colub, NULL);
if (errcode) goto COPT_EXIT;
```

The argument `ncol` specify that the number of variables to create is 3, while the argument `colcost`, `collb` and `colub` specify the costs in objective, lower and upper bound respectively. Regarding other arguments of `COPT_AddCols` for specifing variables types and names, we just pass `NULL` to them, which means all variables are continuous and names are automatically generated by the Cardinal Optimizer. For the remaining arguments, we passed `NULL` too for further action.

Similarly, if non-zero value was returned, it will jump to error reporting code block for detailed information and exit.

Adding constraints

The next step to do after adding variables successfully is to add constraints to problem. For the problem above, the implementation is shown below:

```
// Add constraints
//
//   r0: 1.5 C0 + 1.2 C1 + 1.8 C2 <= 2.6
//   r1: 0.8 C0 + 0.6 C1 + 0.9 C2 >= 1.2
//
int nrow = 2;
int rowbeg[] = {0, 3};
int rowcnt[] = {3, 3};
int rowind[] = {0, 1, 2, 0, 1, 2};
double rowelem[] = {1.5, 1.2, 1.8, 0.8, 0.6, 0.9};
char rowsen[] = {COPT_LESS_EQUAL, COPT_GREATER_EQUAL};
double rowrhs[] = {2.6, 1.2};

errcode = COPT_AddRows(prob, nrow, rowbeg, rowcnt, rowind, rowelem,
                       rowsen, rowrhs, NULL, NULL);
if (errcode) goto COPT_EXIT;
```

The argument `nrow` specifies that the number of constraints to create is 2, while argument `rowbeg`, `rowcnt`, `rowind` and `rowelem` define the coefficient matrix in CSR format. The argument `rowsen` represents the sense of constraints, while argument `rowrhs` specifies the right hand side of constraints. For remaining arguments in `COPT_AddRows`, we simply pass `NULL` to them.

If the return value is non-zero, then it jump to error reporting code block for detailed information and exit.

Setting parameters and attributes

Users are allowed to set parameters and attributes of problem before solving. For example, to set the time limit to 10 seconds, and to set the optimization direction to maximization, the code is shown below:

```
// Set parameters and attributes
errcode = COPT_SetDblParam(prob, COPT_DBLPARAM_TIMELIMIT, 10);
if (errcode) goto COPT_EXIT;
errcode = COPT_SetObjSense(prob, COPT_MAXIMIZE);
if (errcode) goto COPT_EXIT;
```

If non-zero value was returned, then it will jump to error reporting code block for detailed information and exit.

Solve the problem

The next step to do is to solve the problem using code beblow:

```
// Solve problem
errcode = COPT_SolveLp(prob);
if (errcode) goto COPT_EXIT;
```

Non-zero return value indicates unsuccessful solve and jump to error reporting code block for detailed information and exit.

Analyze the solution

Once the solving process was finished, check the solution status first. If it claimed to have found the optimal solution, then use code below to obtain objective value, variables' solution and basis status:

```
// Analyze solution
int lpstat = COPT_LPSTATUS_UNSTARTED;
double lpobjval;
double *lpsol = NULL;
int *colstat = NULL;

errcode = COPT_GetIntAttr(prob, COPT_INTATTR_LPSTATUS, &lpstat);
if (errcode) goto COPT_EXIT;

if (lpstat == COPT_LPSTATUS_OPTIMAL) {
  lpsol = (double *) malloc(ncol * sizeof(double));
  colstat = (int *) malloc(ncol * sizeof(int));

  errcode = COPT_GetLpSolution(prob, lpsol, NULL, NULL, NULL);
  if (errcode) goto COPT_EXIT;

  errcode = COPT_GetBasis(prob, colstat, NULL);
  if (errcode) goto COPT_EXIT;

  errcode = COPT_GetDblAttr(prob, COPT_DBLATTR_LPOBJVAL, &lpobjval);
  if (errcode) goto COPT_EXIT;

  printf("\nObjective value: %.6f\n", lpobjval);

  printf("Variable solution: \n");
  for (int i = 0; i < ncol; ++i)
    printf("  x[%d] = %.6f\n", i, lpsol[i]);

  printf("Variable basis status: \n");
  for (int i = 0; i < ncol; ++i)
    printf("  x[%d]: %d\n", i, colstat[i]);

  free(lpsol);
  free(colstat);
}
```

Write problem and solution

Users are allowed not only to save the problem to solve to standard MPS file, but also the solution, basis status and modified parameters to files:

```
  // Write problem, solution and modified parameters to files
  errcode = COPT_WriteMps(prob, "lp_ex1.mps");
  if (errcode) goto COPT_EXIT;
  errcode = COPT_WriteBasis(prob, "lp_ex1.bas");
  if (errcode) goto COPT_EXIT;
  errcode = COPT_WriteSol(prob, "lp_ex1.sol");
  if (errcode) goto COPT_EXIT;
  errcode = COPT_WriteParam(prob, "lp_ex1.par");
  if (errcode) goto COPT_EXIT;
```

Error handling

The error handling block report error code and message by checking if the return value was non-zero:

```
  // Error handling
COPT_EXIT:
  if (errcode) {
    char errmsg[COPT_BUFFSIZE];

    COPT_GetRetcodeMsg(errcode, errmsg, COPT_BUFFSIZE);
    printf("ERROR %d: %s\n", errcode, errmsg);

    return 0;
  }
```

Delete environment and problem

Before exiting, delete problem and environment respectively:

```
  // Delete problem and environment
  COPT_DeleteProb(&prob);

  COPT_DeleteEnv(&env);
```

6.1.2 Build and run

To ease the work for running the example for users on different operating systems, we provide Visual Studio project and Makefile for Windows, Linux and MacOS respectively, details are shown below.

Windows

For users on Windows platform, we provide Visual Studio project, all users are required to install Visual Studio 2017 beforehand. Assume that the installation directory is: `'<instdir>'`, users that install the Cardinal Optimizer with executable installer can change directory to `'<instdir>\examples\c\vsprojects'` and open the Visual Studio project `lp_ex1.vcxproj` to build the solution. Users that install the Cardinal Optimizer using ZIP-format archive should make sure that all required environment variables are set correctly, see *Install Guide for Cardinal Optimizer* for details.

Linux and MacOS

For users on Linux or MacOS, we provide Makefile to build the example. Please install GCC toolchain for Linux and Clang toolchain for MacOS, together with the `make` utility beforehand. What's more, users should make sure also that all required environment variables are set correctly, see *Install Guide for Cardinal Optimizer* for details. Let's assume that the installation directory of Cardinal Optimizer is `'<instdir>'`, then users need to change directory to `'<instdir>\examples\c'` and execute command `make` in terminal.

6.2 C++ Interface

This chapter walks through a simple C++ example to illustrate the use of the COPT C++ interface. In short words, the example creates an environment, builds a model, add variables and constraints, optimizes it, and then outputs the optimal objective value.

The example solves the following linear problem:

$$\begin{aligned} &\text{Maximize:} \\ &\qquad 1.2x + 1.8y + 2.1z \\ &\text{Subject to:} \\ &\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\ &\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\ &\text{Bounds:} \\ &\qquad 0.1 \leq x \leq 0.6 \\ &\qquad 0.2 \leq y \leq 1.5 \\ &\qquad 0.3 \leq z \leq 2.8 \end{aligned} \tag{6.2}$$

Note that this is the same problem that was modelled and optimized in chapter of *C Interface*.

6.2.1 Example details

Below is the source code solving the above problem using COPT C++ interface.

Listing 6.2: `lp_ex1.cpp`

```
/*
 * This file is part of the Cardinal Optimizer, all rights reserved.
 */
#include "coptcpp_pch.h"

using namespace std;

// The problem to solve:
//
//  Maximize:
//    1.2 x + 1.8 y + 2.1 z
//
//  Subject to:
//    1.5 x + 1.2 y + 1.8 z <= 2.6
//    0.8 x + 0.6 y + 0.9 z >= 1.2
//
//  where:
//    0.1 <= x <= 0.6
//    0.2 <= y <= 1.5
//    0.3 <= z <= 2.8
int main(int argc, char *argv[])
```

(continues on next page)

(continued from previous page)

```
{
  try
  {
    Envr env;
    Model model = env.CreateModel("lp_ex1");

    // Add variables
    //
    //   obj: 1.2 x + 1.8 y + 2.1 z
    //
    //   var:
    //     0.1 <= x <= 0.6
    //     0.2 <= y <= 1.5
    //     0.3 <= z <= 2.8
    //
    Var x = model.AddVar(0.1, 0.6, 0.0, COPT_CONTINUOUS, "x");
    Var y = model.AddVar(0.2, 1.5, 0.0, COPT_CONTINUOUS, "y");
    Var z = model.AddVar(0.3, 2.8, 0.0, COPT_CONTINUOUS, "z");

    model.SetObjective(1.2 * x + 1.8 * y + 2.1 * z, COPT_MAXIMIZE);

    // Add new constraints using linear expression
    //
    //   r0: 1.5 x + 1.2 y + 1.8 z <= 2.6
    //   r1: 0.8 x + 0.6 y + 0.9 z >= 1.2
    //
    model.AddConstr(1.5 * x + 1.2 * y + 1.8 * z <= 2.6, "r0");

    Expr expr(x, 0.8);
    expr.AddTerm(y, 0.6);
    expr += 0.9 * z;
    model.AddConstr(expr >= 1.2, "r1");

    // Set parameters
    model.SetDblParam(COPT_DBLPARAM_TIMELIMIT, 10);

    // Solve problem
    model.Solve();

    // Output solution
    if (model.GetIntAttr(COPT_INTATTR_HASLPSOL) != 0)
    {
      cout << "\nFound optimal solution:" << endl;
      VarArray vars = model.GetVars();
      for (int i = 0; i < vars.Size(); i++)
      {
        Var var = vars.GetVar(i);
        cout << "  " << var.GetName() << " = " << var.Get(COPT_DBLINFO_VALUE) << endl;
      }
      cout << "Obj = " << model.GetDblAttr(COPT_DBLATTR_LPOBJVAL) << endl;
    }
  }
  catch (CoptException e)
  {
    cout << "Error Code = " << e.GetCode() << endl;
    cout << e.what() << endl;
  }
  catch (...)
  {
    cout << "Unknown exception occurs!";
  }
```

(continues on next page)

(continued from previous page)

```
}
```

Let's now walk through the example, line by line, to understand how it achieves the desired result of optimizing the model. Note that the example must include header coptcpp_pch.h.

Creating environment and model

Essentially, any C++ application using Cardinal Optimizer should start with a COPT environment, where user could add one or more models. Note that each model encapsulates a problem and corresponding data.

Furthermore, to create multiple problems, one can load them one by one in the same model, besides the naive option of creating multiple models in the environment.

```
Envr env;
Model model = env.CreateModel("lp_ex1");
```

The above call instantiates a COPT environment and a model with name "lp_ex1".

Adding variables

The next step in our example is to add variables to the model. Variables are added through AddVar() or AddVars() method on the model object. A variable is always associated with a particular model.

```
// Add variables
//
//   obj: 1.2 x + 1.8 y + 2.1 z
//
//   var:
//     0.1 <= x <= 0.6
//     0.2 <= y <= 1.5
//     0.3 <= z <= 2.8
//
Var x = model.AddVar(0.1, 0.6, 0.0, COPT_CONTINUOUS, "x");
Var y = model.AddVar(0.2, 1.5, 0.0, COPT_CONTINUOUS, "y");
Var z = model.AddVar(0.3, 2.8, 0.0, COPT_CONTINUOUS, "z");

model.SetObjective(1.2 * x + 1.8 * y + 2.1 * z, COPT_MAXIMIZE);
```

The first and second arguments to the AddVar() call are the variable lower and upper bounds, respectively. The third argument is the linear objective coefficient (zero here - we'll set the objective later). The fourth argument is the variable type. Our variables are all continuous in this example. The final argument is the name of the variable.

The AddVar() method has been overloaded to accept several different argument lists. Please refer to *C++ API Reference* for further details.

The objective is built here using overloaded operators. The C++ API overloads the arithmetic operators to allow you to build linear expressions by COPT variables. The second argument indicates that the sense is maximization.

Adding constraints

The next step in the example is to add the linear constraints. As with variables, constraints are always associated with a specific model. They are created using AddConstr() or AddConstrs() methods on the model object.

```
// Add new constraints using linear expression
//
//   r0: 1.5 x + 1.2 y + 1.8 z <= 2.6
//   r1: 0.8 x + 0.6 y + 0.9 z >= 1.2
//
model.AddConstr(1.5 * x + 1.2 * y + 1.8 * z <= 2.6, "r0");

Expr expr(x, 0.8);
expr.AddTerm(y, 0.6);
expr += 0.9 * z;
model.AddConstr(expr >= 1.2, "r1");
```

The first constraint is to use overloaded arithmetic operators to build the linear expression. The comparison operators are also overloaded to make it easier to build constraints.

The second constraint is created by building a linear expression incrementally. That is, an expression can be built by constructor of a variable and its coefficient, by AddTerm() method, and by overloaded operators.

Setting parameters and attributes

The next step in the example is to set parameters and attributes of the problem before optimization.

```
// Set parameters
model.SetDblParam(COPT_DBLPARAM_TIMELIMIT, 10);
```

The SetDblParam() call here with COPT_DBLPARAM_TIMELIMIT argument sets solver to optimize up to 10 seconds.

Solving problem

Now that the model has been built, the next step is to optimize it:

```
// Solve problem
model.Solve();
```

This routine performs the optimization and populates several internal model attributes (including the status of the optimization, the solution, etc.).

Outputting solution

After solving the problem, one can query the values of the attributes for various of purposes.

```
// Output solution
if (model.GetIntAttr(COPT_INTATTR_HASLPSOL) != 0)
{
  cout << "\nFound optimal solution:" << endl;
  VarArray vars = model.GetVars();
  for (int i = 0; i < vars.Size(); i++)
  {
    Var var = vars.GetVar(i);
    cout << "  " << var.GetName() << " = " << var.Get(COPT_DBLINFO_VALUE) << endl;
```

(continues on next page)

(continued from previous page)

```
    }
    cout << "Obj = " << model.GetDblAttr(COPT_DBLATTR_LPOBJVAL) << endl;
  }
```

Spcifically, one can query the COPT_INTATTR_HASLPSOL attribute on the model to know whether we have optimal LP solution; query the COPT_DBLINFO_VALUE attribute of a variable to obtain its solution value; query the COPT_DBLATTR_LPOBJVAL attribute on the model to obtain the objective value for the current solution.

The names and types of all model, variable, and constraint attributes can be found in *Attributes* of C API reference.

Error handling

Errors in the COPT C++ interface are handled through the C++ exception mechanism. In the example, all COPT statements are enclosed inside a try block, and any associated errors would be caught by the catch block.

```
catch (CoptException e)
{
  cout << "Error Code = " << e.GetCode() << endl;
  cout << e.what() << endl;
}
catch (...)
{
  cout << "Unknown exception occurs!";
}
```

6.2.2 Build and Run

To build and run the example, users may refer to files under `$COPT_HOME/examples/cpp`. Specifically, We provide visual studio project on Windows, as well as makefile project on Linux and Mac platforms.

Windows Visual Studio project

For Windows platform, Visual Studio project is located at `$COPT_HOME/examples/cpp/vsprojects`. Double-clicking the project file `lp_ex1.vcxproj` will bring Visual Studio. Note that it requires Visual studio 2017 or 2019 installed on Windows 10 to build.

The Visual Studio project has dependency on COPT cpp shared libary `copt_cpp.dll`, which is refered in project file, along with its import library `copt_cpp.lib`. The required headers are `copt.h`, `coptcpp.h` and `coptcpp.idl.h`, which declare COPT constants, interfaces and methods exported from `copt_cpp.dll`. In addition, the example provides class header files under `$COPT_HOME/include/coptcpp_inc`, which wraps COPT cpp interfaces and redefines overloaded operators. For simplicity, user should include `coptcpp_pch.h` to start with, as shown in the example.

To run the example, users should have COPT installed. Specifically, it requires COPT cpp libary, `copt_cpp.dll`, and valid license files to run. Please refer to *Install Guide for Cardinal Optimizer* for further details.

It is `IMPORTANT` to notice that COPT cpp shared library, `copt_cpp.dll`, is not compatible with gcc compiler on Windows. That is, if you are running this example on Windows, the executable compiled by gcc will not work as expected. This is because GCC is not compabile with Windows SDK. On the other hand, both Clang and Intel compiler work fine, as well as MSVC.

Makefile project

We provide Makefile to build the example for Linux and MacOS platforms. Please make sure tools, `gcc` and `make`, are already installed on the platforms. To build the example, change directory to `$COPT_HOME/examples/cpp` and execute command `'make'` on unix terminal.

The project has dependency on COPT cpp shared libary, that is, `libcopt_cpp.so` on Linux platform and `libcopt_cpp.dylib` on MacOS. Similar to Windows VS project, user should refer to `coptcpp_pch.h` under `$COPT_HOME/include/coptcpp_inc'` to include all necessary headers, as shown in the example.

To run the example, users should have COPT installed. Specifically, it requires COPT cpp libary, `libcopt_cpp.so` on Linux, `libcopt_cpp.dylib` on MacOS, and valid license files to run. Alternatively, user might set `LD_LIBRARY_PATH` and `COPT_LICENSE_DIR` properly to work around. Please refer to *Install Guide for Cardinal Optimizer* for further details.

6.3 C# Interface

This chapter walks through a simple C# example to illustrate the use of the COPT C# interface. In short words, the example creates an environment, builds a model, add variables and constraints, optimizes it, and then outputs the optimal objective value.

The example solves the following linear problem:

$$\begin{aligned} &\text{Maximize:} \\ &\qquad 1.2x + 1.8y + 2.1z \\ &\text{Subject to:} \\ &\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\ &\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\ &\text{Bounds:} \\ &\qquad 0.1 \leq x \leq 0.6 \\ &\qquad 0.2 \leq y \leq 1.5 \\ &\qquad 0.3 \leq z \leq 2.8 \end{aligned} \tag{6.3}$$

Note that this is the same problem that was modelled and optimized in chapter of *C Interface*.

6.3.1 Example details

Below is the source code solving the above problem using COPT C# interface.

Listing 6.3: `lp_ex1.cs`

```
/*
 * This file is part of the Cardinal Optimizer, all rights reserved.
 */
using Copt;
using System;

// The problem to solve:
//
//  Maximize:
//    1.2 x + 1.8 y + 2.1 z
//
//  Subject to:
//    1.5 x + 1.2 y + 1.8 z <= 2.6
//    0.8 x + 0.6 y + 0.9 z >= 1.2
//
```

(continues on next page)

(continued from previous page)

```
//  where:
//    0.1 <= x <= 0.6
//    0.2 <= y <= 1.5
//    0.3 <= z <= 2.8
public class lp_ex1
{
  public static void Main()
  {
    try
    {
      Envr env = new Envr();
      Model model = env.CreateModel("lp_ex1");

      // Add variables
      //
      //   obj: 1.2 x + 1.8 y + 2.1 z
      //
      //   var:
      //     0.1 <= x <= 0.6
      //     0.2 <= y <= 1.5
      //     0.3 <= z <= 2.8
      //
      Var x = model.AddVar(0.1, 0.6, 0.0, Copt.Consts.CONTINUOUS, "x");
      Var y = model.AddVar(0.2, 1.5, 0.0, Copt.Consts.CONTINUOUS, "y");
      Var z = model.AddVar(0.3, 2.8, 0.0, Copt.Consts.CONTINUOUS, "z");

      model.SetObjective(1.2 * x + 1.8 * y + 2.1 * z, Copt.Consts.MAXIMIZE);

      // Add new constraints using linear expression
      //
      //   r0: 1.5 x + 1.2 y + 1.8 z <= 2.6
      //   r1: 0.8 x + 0.6 y + 0.9 z >= 1.2
      //
      model.AddConstr(1.5 * x + 1.2 * y + 1.8 * z <= 2.6, "r0");

      Expr expr = new Expr(x, 0.8);
      expr.AddTerm(y, 0.6);
      expr += 0.9 * z;
      model.AddConstr(expr >= 1.2, "r1");

      // Set parameters
      model.SetDblParam(Copt.DblParam.TimeLimit, 10);

      // Solve problem
      model.Solve();

      // Output solution
      if (model.GetIntAttr(Copt.IntAttr.LpStatus) == Copt.Status.OPTIMAL)
      {
        Console.WriteLine("\nFound optimal solution:");
        VarArray vars = model.GetVars();
        for (int i = 0; i < vars.Size(); i++)
        {
          Var var = vars.GetVar(i);
          Console.WriteLine("  {0} = {1}", var.GetName(),
                                           var.Get(Copt.DblInfo.Value));
        }
        Console.WriteLine("Obj = {0}", model.GetDblAttr(Copt.DblAttr.LpObjVal));
      }
      Console.WriteLine("\nDone");
    }
```

(continues on next page)

(continued from previous page)

```
77      catch (CoptException e)
78      {
79        Console.WriteLine("Error Code = {0}", e.GetCode());
80        Console.WriteLine(e.Message);
81      }
82    }
83  }
```

Let's now walk through the example, line by line, to understand how it achieves the desired result of optimizing the model.

Creating environment and model

Essentially, any C# application using Cardinal Optimizer should start with a COPT environment, where user could add one or more models. Note that each model encapsulates a problem and corresponding data.

Furthermore, to create multiple problems, one can load them one by one in the same model, besides the naive option of creating multiple models in the environment.

```
      Envr env = new Envr();
      Model model = env.CreateModel("lp_ex1");
```

The above call instantiates a COPT environment and a model with name "lp_ex1".

Adding variables

The next step in our example is to add variables to the model. Variables are added through AddVar() or AddVars() method on the model object. A variable is always associated with a particular model.

```
      // Add variables
      //
      //   obj: 1.2 x + 1.8 y + 2.1 z
      //
      //   var:
      //     0.1 <= x <= 0.6
      //     0.2 <= y <= 1.5
      //     0.3 <= z <= 2.8
      //
      Var x = model.AddVar(0.1, 0.6, 0.0, Copt.Consts.CONTINUOUS, "x");
      Var y = model.AddVar(0.2, 1.5, 0.0, Copt.Consts.CONTINUOUS, "y");
      Var z = model.AddVar(0.3, 2.8, 0.0, Copt.Consts.CONTINUOUS, "z");

      model.SetObjective(1.2 * x + 1.8 * y + 2.1 * z, Copt.Consts.MAXIMIZE);
```

The first and second arguments to the AddVar() call are the variable lower and upper bounds, respectively. The third argument is the linear objective coefficient (zero here - we'll set the objective later). The fourth argument is the variable type. Our variables are all continuous in this example. The final argument is the name of the variable.

The AddVar() method has been overloaded to accept several different argument lists. Please refer to chapCSharpApiRef_Class of C# API reference for further details.

The objective is built here using overloaded operators. The C# API overloads the arithmetic operators to allow you to build linear expressions by COPT variables. The second argument indicates that the sense is maximization.

Adding constraints

The next step in the example is to add the linear constraints. As with variables, constraints are always associated with a specific model. They are created using AddConstr() or AddConstrs() methods on the model object.

```
// Add new constraints using linear expression
//
//   r0: 1.5 x + 1.2 y + 1.8 z <= 2.6
//   r1: 0.8 x + 0.6 y + 0.9 z >= 1.2
//
model.AddConstr(1.5 * x + 1.2 * y + 1.8 * z <= 2.6, "r0");

Expr expr = new Expr(x, 0.8);
expr.AddTerm(y, 0.6);
expr += 0.9 * z;
model.AddConstr(expr >= 1.2, "r1");
```

The first constraint is to use overloaded arithmetic operators to build the linear expression. The comparison operators are also overloaded to make it easier to build constraints.

The second constraint is created by building a linear expression incrementally. That is, an expression can be built by constructor of a variable and its coefficient, by AddTerm(), and by overloaded operators.

Setting parameters and attributes

The next step in the example is to set parameters and attributes of the problem before optimization.

```
// Set parameters
model.SetDblParam(Copt.DblParam.TimeLimit, 10);
```

The SetDblParam() call here with Copt.DblParam.TimeLimit argument sets solver to optimize up to 10 seconds.

Solving problem

Now that the model has been built, the next step is to optimize it:

```
// Solve problem
model.Solve();
```

This routine performs the optimization and populates several internal model attributes (including the status of the optimization, the solution, etc.).

Outputting solution

After solving the problem, one can query the values of the attributes for various of purposes.

```
// Output solution
if (model.GetIntAttr(Copt.IntAttr.LpStatus) == Copt.Status.OPTIMAL)
{
  Console.WriteLine("\nFound optimal solution:");
  VarArray vars = model.GetVars();
  for (int i = 0; i < vars.Size(); i++)
  {
    Var var = vars.GetVar(i);
    Console.WriteLine("  {0} = {1}", var.GetName(),
                                     var.Get(Copt.DblInfo.Value));
  }
```

(continues on next page)

(continued from previous page)

```
        Console.WriteLine("Obj = {0}", model.GetDblAttr(Copt.DblAttr.LpObjVal));
      }
      Console.WriteLine("\nDone");
```

Spcifically, one can query the Copt.IntAttr.LpStatus attribute of the model to determine whether we have found optimal LP solution; query the Copt.DblInfo.Value attribute of a variable to obtain its solution value; query the Copt.DblAttr.LpObjVal attribute on the model to obtain the objective value for the current solution.

The names and types of all model, variable, and constraint attributes can be found in *Constants* of C# API reference.

Error handling

Errors in the COPT C# interface are handled through the C# exception mechanism. In the example, all COPT statements are enclosed inside a try block, and any associated errors would be caught by the catch block.

```
      Console.WriteLine("Error Code = {0}", e.GetCode());
      Console.WriteLine(e.Message);
    }
  }
}
```

6.3.2 Build and Run

To build and run csharp example, users may refer to project under `$COPT_HOME/examples/csharp`. Specifically, We provide a csharp project file in cross-platform framework of dotnet core 2.0. This example shows a single project working on Windows, as well as Linux and Mac platforms.

First of all, download and install dotnet core 2.0 on your platform. To get started, follow instructions in the dotnet core docs.

Dotnet core 2.0 project

The dotnet core 2.0 project file `example.csproj` example locates in folder `$COPT_HOME/examples/csharp/dotnetprojects`. Copy example file `lp_ex1.cs` to this folder and change directory to there by Windows command line prompt, then run with command `'dotnet run --framework netcoreapp2.0'`. For users of dotnet core 3.0, just run with `'dotnet run --framework netcoreapp3.0'` instead will work too.

This csharp project has dependency on COPT dotnet 2.0 shared libary `copt_dotnet20.dll`, which is refered in the project file and defines all managed classes of COPT solver. In addition, `copt_dotnet20.dll` loads two shared libraries, that is, `coptcswrap.dll` and `copt_cpp.dll` on Windows, `libcoptcswrap.so` and `libcopt_cpp.so` on Linux, `libcoptcswrap.dylib` and `libcopt_cpp.dylib` on Mac respectively. Note that `coptcswrap` library is a bridge between managed COPT library and native library `copt_cpp.dll`, which declares and implements COPT constants, interfaces and methods. So users should make sure they are installed properly on runtime search paths.

In summary, to run csharp example, users should have COPT installed properly. Specifically, it requires three related COPT shared libaries existing on runtime search paths, and valid license files to run. Please refer to *Install Guide for Cardinal Optimizer* for further details.

6.4 Java Interface

This chapter walks through a simple Java example to illustrate the use of the COPT Java interface. In short words, the example creates an environment, builds a model, add variables and constraints, optimizes it, and then outputs the optimal objective value.

The example solves the following linear problem:

$$\begin{aligned} &\text{Maximize:} \\ &\qquad 1.2x + 1.8y + 2.1z \\ &\text{Subject to:} \\ &\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\ &\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\ &\text{Bounds:} \\ &\qquad 0.1 \leq x \leq 0.6 \\ &\qquad 0.2 \leq y \leq 1.5 \\ &\qquad 0.3 \leq z \leq 2.8 \end{aligned} \tag{6.4}$$

Note that this is the same problem that was modelled and optimized in chapter of *C Interface*.

6.4.1 Example details

Below is the source code solving the above problem using COPT Java interface.

Listing 6.4: `Lp_ex1.java`

```
/*
 * This file is part of the Cardinal Optimizer, all rights reserved.
 */
import copt.*;

// The problem to solve:
//
//  Maximize:
//    1.2 x + 1.8 y + 2.1 z
//
//  Subject to:
//    1.5 x + 1.2 y + 1.8 z <= 2.6
//    0.8 x + 0.6 y + 0.9 z >= 1.2
//
//  where:
//    0.1 <= x <= 0.6
//    0.2 <= y <= 1.5
//    0.3 <= z <= 2.8
public class Lp_ex1 {
  public static void main(final String argv[]) {
    try {
      Envr env = new Envr();
      Model model = env.createModel("lp_ex1");

      // Add variables
      //
      // obj: 1.2 x + 1.8 y + 2.1 z
      //
      // var:
      // 0.1 <= x <= 0.6
      // 0.2 <= y <= 1.5
      // 0.3 <= z <= 2.8
```

(continues on next page)

(continued from previous page)

```
      //
      Var x = model.addVar(0.1, 0.6, 1.2, copt.Consts.CONTINUOUS, "x");
      Var y = model.addVar(0.2, 1.5, 1.8, copt.Consts.CONTINUOUS, "y");
      Var z = model.addVar(0.3, 2.8, 2.1, copt.Consts.CONTINUOUS, "z");

      // Add new constraints using linear expression
      //
      // r0: 1.5 x + 1.2 y + 1.8 z <= 2.6
      // r1: 0.8 x + 0.6 y + 0.9 z >= 1.2
      //
      Expr e0 = new Expr(x, 1.5);
      e0.addTerm(y, 1.2);
      e0.addTerm(z, 1.8);
      model.addConstr(e0, copt.Consts.LESS_EQUAL, 2.6, "r0");

      Expr e1 = new Expr(x, 0.8);
      e1.addTerm(y, 0.6);
      e1.addTerm(z, 0.9);
      model.addConstr(e1, copt.Consts.GREATER_EQUAL, 1.2, "r1");

      // Set parameters and attributes
      model.setDblParam(copt.DblParam.TimeLimit, 10);
      model.setObjSense(copt.Consts.MAXIMIZE);

      // Solve problem
      model.solve();

      // Output solution
      if (model.getIntAttr(copt.IntAttr.HasLpSol) != 0) {
        System.out.println("\nFound optimal solution:");
        VarArray vars = model.getVars();
        for (int i = 0; i < vars.size(); i++) {
          Var var = vars.getVar(i);
          System.out.println("  " + var.getName() + " = "
                             + var.get(copt.DblInfo.Value));
        }
        System.out.println("Obj = " + model.getDblAttr(copt.DblAttr.LpObjVal));
      }

      System.out.println("\nDone");
    } catch (CoptException e) {
      System.out.println("Error Code = " + e.getCode());
      System.out.println(e.getMessage());
    }
  }
}
```

Let's now walk through the example, line by line, to understand how it achieves the desired result of optimizing the model.

Import COPT class

To use the Java interface of COPT, users need to import the Java interface class of COPT first.

```
import copt.*;
```

Creating environment and model

Essentially, any Java application using Cardinal Optimizer should start with a COPT environment, where user could add one or more models. Note that each model encapsulates a problem and corresponding data.

Furthermore, to create multiple problems, one can load them one by one in the same model, besides the naive option of creating multiple models in the environment.

```
    Envr env = new Envr();
    Model model = env.createModel("lp_ex1");
```

The above call instantiates a COPT environment and a model with name "COPT Java Example".

Adding variables

The next step in our example is to add variables to the model. Variables are added through addVar() or addVars() method on the model object. A variable is always associated with a particular model.

```
    // Add variables
    //
    // obj: 1.2 x + 1.8 y + 2.1 z
    //
    // var:
    // 0.1 <= x <= 0.6
    // 0.2 <= y <= 1.5
    // 0.3 <= z <= 2.8
    //
    Var x = model.addVar(0.1, 0.6, 1.2, copt.Consts.CONTINUOUS, "x");
    Var y = model.addVar(0.2, 1.5, 1.8, copt.Consts.CONTINUOUS, "y");
    Var z = model.addVar(0.3, 2.8, 2.1, copt.Consts.CONTINUOUS, "z");
```

The first and second arguments to the addVar() call are the variable lower and upper bounds, respectively. The third argument is the linear objective coefficient. The fourth argument is the variable type. Our variables are all continuous in this example. The final argument is the name of the variable.

The addVar() method has been overloaded to accept several different argument lists. Please refer to *Java Modeling Classes* of Java API reference for further details.

Adding constraints

The next step in the example is to add the linear constraints. As with variables, constraints are always associated with a specific model. They are created using addConstr() or addConstrs() methods on the model object.

```
    // Add new constraints using linear expression
    //
    // r0: 1.5 x + 1.2 y + 1.8 z <= 2.6
    // r1: 0.8 x + 0.6 y + 0.9 z >= 1.2
    //
    Expr e0 = new Expr(x, 1.5);
    e0.addTerm(y, 1.2);
```

(continues on next page)

(continued from previous page)

```
e0.addTerm(z, 1.8);
model.addConstr(e0, copt.Consts.LESS_EQUAL, 2.6, "r0");

Expr e1 = new Expr(x, 0.8);
e1.addTerm(y, 0.6);
e1.addTerm(z, 0.9);
model.addConstr(e1, copt.Consts.GREATER_EQUAL, 1.2, "r1");
```

Two constraints here are created by building linear expressions incrementally. That is, an expression can be built by constructor of a variable and its coefficient, and then by addTerm() method.

Setting parameters and attributes

The next step in the example is to set parameters and attributes of the problem before optimization.

```
// Set parameters and attributes
model.setDblParam(copt.DblParam.TimeLimit, 10);
model.setObjSense(copt.Consts.MAXIMIZE);
```

The setDblParam() call here with copt.DblParam.TimeLimit argument sets solver to optimize up to 10 seconds. The setObjSense() call with copt.Consts.MAXIMIZE argument sets objective sense as maximization.

Solving problem

Now that the model has been built, the next step is to optimize it:

```
// Solve problem
model.solve();
```

This routine performs the optimization and populates several internal model attributes (including the status of the optimization, the solution, etc.).

Outputting solution

After solving the problem, one can query the values of the attributes for various of purposes.

```
// Output solution
if (model.getIntAttr(copt.IntAttr.HasLpSol) != 0) {
  System.out.println("\nFound optimal solution:");
  VarArray vars = model.getVars();
  for (int i = 0; i < vars.size(); i++) {
    Var var = vars.getVar(i);
    System.out.println("  " + var.getName() + " = "
                       + var.get(copt.DblInfo.Value));
  }
  System.out.println("Obj = " + model.getDblAttr(copt.DblAttr.LpObjVal));
}

System.out.println("\nDone");
```

Spcifically, one can query the copt.IntAttr.HasLpSol attribute on the model to know whether we have optimal LP solution; query the copt.DblInfo.Value attribute of a variable to obtain its solution value; query the copt.DblAttr.LpObjVal attribute on the model to obtain the objective value for the current solution.

The names and types of all model, variable, and constraint attributes can be found in *Constants* of Java API reference.

Error handling

Errors in the COPT Java interface are handled through the Java exception mechanism. In the example, all COPT statements are enclosed inside a try block, and any associated errors would be caught by the catch block.

```
    } catch (CoptException e) {
      System.out.println("Error Code = " + e.getCode());
      System.out.println(e.getMessage());
    }
```

6.4.2 Build and Run

To build and run java example, users may refer to files under `$COPT_HOME/examples/java`. Specifically, We provide an example file in java and a script file to build. This single example runs on all platforms that support Java.

First of all, download and install *Java 8 or above* on your platform.

Java example detail

In the java example folder `$COPT_HOME/examples/java`, the easiest way to run the example is to enter the java example folder in console or terminal and then execute command `'sh run.sh'`.

This java project has dependency on COPT java package `copt_java.jar`, which defines all java classes of COPT solver. In addition, `copt_java.jar` loads two shared libraries, that is, `coptjniwrap.dll` and `copt_cpp.dll` on Windows, `libcoptjniwrap.so` and `libcopt_cpp.so` on Linux, `libcoptjniwrap.dylib` and `libcopt_cpp.dylib` on Mac respectively. Note that `coptjniwrap` library is a JNI swig wrapper and acts as a bridge between COPT java package and native library `copt_cpp`, which declares and implements COPT constants, interfaces and methods. So users should make sure they are installed properly on runtime search paths.

In summary, to run java example, users should have COPT installed properly. Specifically, it requires two related COPT shared libaries existing on runtime search paths, and valid license files to run. Please refer to *Install Guide for Cardinal Optimizer* for further details.

6.5 Python Interface

6.5.1 Installation guide

Currently, the Python interface of the Cardinal Optimizer (COPT) supports Python 2.7, 3.7-3.11 versions (for MacOS M1, COPT supports Python 3.8-3.11 versions). Before using the Python interface, please ensure that COPT has been installed and configured correctly. For details, please refer to *How to install the Cardinal Optimizer*. Users can download Python from Anaconda distribution or Python official distribution . We recommend users install the Anaconda distribution, because it is more user-friendly and convenient for Python novices (For Windows, please don't install Python via Microsoft Store). If you use official Python distribution or Python shipped with system, then make sure you have installed the `pip` and `setuptools` Python packages beforehand.

Windows

Method 1: via pip install (recommended)

Open the cmd command window (if Python is an Anaconda distribution, open the Anaconda command line window) and enter the following command:

```
pip install coptpy
```

If an older version of the `coptpy` package has been installed, please open the cmd command window (if Python is an Anaconda distribution, open the Anaconda command line window) and enter the following command to upgrade to the latest version of the `coptpy` package:

```
pip install --upgrade coptpy
```

Method 2: via COPT installation package

For Windows, assuming the installation path of COPT is: `"C:\Program Files\COPT"`, please switch to the directory `"C:\Program Files\COPT\lib\python"` and execute the following commands on command line:

```
python setup.py install
```

Note that if COPT is installed on the system disk, you need to **execute with administrator privileges** to open the command prompt. To test whether the Python interface is installed correctly, users can switch to the directory `"C:\Program Files\COPT\examples\python"` and execute the following commands on the command line:

```
python lp_ex1.py
```

If the model is solved correctly, it means that the Python interface of COPT has been installed correctly.

Note If you use the official release of Python 3.8, assume that its installation path is: `"C:\Program Files\Python38"`, you need to copy the `copt_cpp.dll` file in the `"bin"` subdirectory of the COPT installation path to `"C:\Program Files\Python38\Lib\site-packages\coptpy"` to solve the problem of dynamic library dependency.

Currently, `coptpy` already supports type hints, users can execute the following command in the command line window (if Python is the Anaconda release version, please open Anaconda Prompt):

```
pip install coptpy-stubs
```

After successfully installed, when writing code in the Python IDE, you will be prompted to complete the variable name and the value of the function parameter.

Linux

Method 1: via pip install (recommended)

Open the terminal and enter the following command:

```
pip install coptpy
```

If an older version of the `coptpy` package has been installed, please open the terminal and enter the following command to upgrade to the latest version of the `coptpy` package:

```
pip install --upgrade coptpy
```

Method 2: via COPT installation package

For Linux, suppose the installation path of COPT is: /opt/copt65, please switch to the directory /opt/copt65/lib/python and execute the following commands on terminal:

```
sudo python setup.py install
```

For users using Python from Anaconda distribution, if above commands fails, assuming the installation path of Anaconda is: `"/opt/anaconda3"`, please execute the following commands instead on terminal to install the Python interface of COPT:

```
sudo /opt/anaconda3/bin/python setup.py install
```

To test whether the Python interface is installed correctly, users can switch to the directory /opt/copt65/examples/python and execute the following commands on terminal:

```
python lp_ex1.py
```

If the model solved correctly, it means that the Python interface of COPT has been installed correctly.

Currently, `coptpy` already supports type hints, users can execute the following command in the terminal:

```
pip install coptpy-stubs
```

After successfully installed, when writing code in the Python IDE, you will be prompted to complete the variable name and the value of the function parameter.

MacOS

Method 1: via pip install (recommended)

Open the terminal and enter the following command:

```
pip install coptpy
```

If an older version of the `coptpy` package has been installed, please open the terminal and enter the following command to upgrade to the latest version of the `coptpy` package:

```
pip install --upgrade coptpy
```

Method 2: via COPT installation package

For MacOS, assuming that the installation path of COPT is: /Applications/copt65, please switch the directory to /Applications/copt65/lib/python and execute the following commands on terminal:

```
sudo python setup.py install
```

To test whether the Python interface is installed correctly, users can switch to the directory /Applications/copt65/examples/python and execute the following commands on terminal:

```
python lp_ex1.py
```

If the model solved correctly, it means that the Python interface of COPT has been installed correctly.

Currently, `coptpy` already supports type hints, users can execute the following command in the terminal:

```
pip install coptpy-stubs
```

After successfully installed, when writing code in the Python IDE, you will be prompted to complete the variable name and the value of the function parameter.

6.5.2 Example details

This chapter illustrate the use of C interface of Cardinal Optimizer through a simple Python example. The problem to solve is shown in Eq. 6.5:

$$\begin{aligned}
&\text{Maximize:} \\
&\qquad 1.2x + 1.8y + 2.1z \\
&\text{Subject to:} \\
&\qquad 1.5x + 1.2y + 1.8z \leq 2.6 \\
&\qquad 0.8x + 0.6y + 0.9z \geq 1.2 \\
&\text{Bounds:} \\
&\qquad 0.1 \leq x \leq 0.6 \\
&\qquad 0.2 \leq y \leq 1.5 \\
&\qquad 0.3 \leq z \leq 2.8
\end{aligned} \tag{6.5}$$

The source code for solving the above problem using Python API of Cardinal Optimizer is shown in Listing 6.5:

Listing 6.5: `lp_ex1.py`

```
#
# This file is part of the Cardinal Optimizer, all rights reserved.
#

"""
The problem to solve:

  Maximize:
    1.2 x + 1.8 y + 2.1 z

  Subject to:
    1.5 x + 1.2 y + 1.8 z <= 2.6
    0.8 x + 0.6 y + 0.9 z >= 1.2

  where:
    0.1 <= x <= 0.6
    0.2 <= y <= 1.5
    0.3 <= z <= 2.8
"""

import coptpy as cp
from coptpy import COPT

# Create COPT environment
env = cp.Envr()

# Create COPT model
model = env.createModel("lp_ex1")

# Add variables: x, y, z
x = model.addVar(lb=0.1, ub=0.6, name="x")
y = model.addVar(lb=0.2, ub=1.5, name="y")
z = model.addVar(lb=0.3, ub=2.8, name="z")

# Add constraints
model.addConstr(1.5*x + 1.2*y + 1.8*z <= 2.6)
model.addConstr(0.8*x + 0.6*y + 0.9*z >= 1.2)

# Set objective function
model.setObjective(1.2*x + 1.8*y + 2.1*z, sense=COPT.MAXIMIZE)
```

(continues on next page)

(continued from previous page)

```

# Set parameter
model.setParam(COPT.Param.TimeLimit, 10.0)

# Solve the model
model.solve()

# Analyze solution
if model.status == COPT.OPTIMAL:
  print("Objective value: {}".format(model.objval))
  allvars = model.getVars()

  print("Variable solution:")
  for var in allvars:
    print(" x[{0}]: {1}".format(var.index, var.x))

  print("Variable basis status:")
  for var in allvars:
    print(" x[{0}]: {1}".format(var.index, var.basis))

  # Write model, solution and modified parameters to file
  model.write("lp_ex1.mps")
  model.write("lp_ex1.bas")
  model.write("lp_ex1.sol")
  model.write("lp_ex1.par")
```

We will explain how to use the Python API step by step based on code above, please refer to *C API Reference* for detailed usage of Python API.

Import Python interface

To use the Python interface of COPT, users need to import the Python interface library first.

```
import coptpy as cp
```

Create environment

To solve any problem with COPT, users need to create optimization environment before creating any model.

```

# Create COPT environment
```

Create model

If the optimization environment was created successfully, users need to create the model to solve, which includes variables and constraints information.

```

# Create COPT model
```

Add variables

Users can specify information such as objective costs, lower and upper bounds of variables when creating them. In this example, we just set the lower and upper bounds of variables and their names.

```
# Add variables: x, y, z
x = model.addVar(lb=0.1, ub=0.6, name="x")
y = model.addVar(lb=0.2, ub=1.5, name="y")
```

Add constraints

After adding variables, we can then add constraints to the model.

```
# Add constraints
model.addConstr(1.5*x + 1.2*y + 1.8*z <= 2.6)
```

Set objective function

After adding variables and constraints, we can further specify objective function for the model.

```
# Set objective function
```

Set parameters

Users can set optimization parameters before solving the model, e.g. set optimization time limit to 10 seconds.

```
# Set parameter
```

Solve model

Solve the model via `solve` method.

```
# Solve the model
```

Analyze solution

When solving finished, we should query the optimization status first. If the optimization status is optimal, then we can retrieve objective value, solution and basis status of variables.

```
# Analyze solution
if model.status == COPT.OPTIMAL:
  print("Objective value: {}".format(model.objval))
  allvars = model.getVars()

  print("Variable solution:")
  for var in allvars:
    print(" x[{0}]: {1}".format(var.index, var.x))
```

(continues on next page)

(continued from previous page)

```
print("Variable basis status:")
for var in allvars:
```

Write files

Users can write current model to MPS format file, and write solution, basis status and modified parameters to file.

```
# Write model, solution and modified parameters to file
model.write("lp_ex1.mps")
model.write("lp_ex1.bas")
model.write("lp_ex1.sol")
```

6.5.3 Best Practice

Upgrade to newer version

If users have COPT python installed and need to upgrade latest version, it is recommended to remove previous version before installing new version. To remove previous version, it is as simple as deleting the folder `coptpy` at `site-package`.

Multi-Thread Programming

COPT does not guarrantee thread safe and modelling APIs are not reentrant in general. It is safe to share COPT Envr objects among threads. However, it is not recommended to share Model objects among threads, unless you understand what you are doing. For instance, if you share the same model between two threads. One thread is responsible for modelling and solving. The other thread is used to monitor the progress and may interrupt at some circumstances, such as running out of time.

Dictionary order guarranteed after Python v3.7

As you know, Python dictionaries did not preserve the order in which items were added to them. For instance, you might type `{'fruits': ['apple', 'orange'], 'veggies': ['carrot', 'pea']}` and get back `{'veggies': ['carrot', 'pea'], 'fruits': ['apple', 'orange']}`.

However, the situation is changed. Standard dict objects preserve order in the implementation of Python 3.6. This order-preserving property is becoming a language feature in Python 3.7.

If your program has dependency on dictionary orders, install coptpy for Python v3.7 or later version. For instance, if your model is implemented in Python 2.7 as follows:

```
m = Envr().createModel("customized model")
vx = m.addVars(['hello', 'world'], [0, 1, 2], nameprefix = "X")
# add a constraint for each var in tupledict 'vx'
m.addConstrs(vx[key] >= 1.0 for key in vx)
```

Your model might end up with rows `{R(hello,1), R(hello,0), R(world,1), R(world,0), R(hello,2), R(world,2)}` .

Use quicksum and psdquicksum when possible

COPT pythyon package supports building linear expression, quadratic expression and PSD expression in natural way. For linear and quadratic expression, it is recommended to use quicksum() to build expression objects. For linear and PSD expression, it is recommended to use psdquicksum() to build expression objects. Both of them implement inplace summation, which is much faster than standard plus operator.

6.6 AMPL Interface

AMPL is an algebraic modeling language for describing large-scale complex mathematical problems, it was hooked to many commercial and open-source mathematical optimizers, with various data interfaces and extensions, and received high popularity among both industries and institutes, see Who uses AMPL? for more information. The solver `coptampl` uses **Cardinal Optimizer** to solve linear programming, convex quadratic programming, convex quadratic constrainted programming and mixed integer programming problems. Normally `coptampl` is invoked by AMPL's solve command, which gives the invocation:

```
coptampl stub -AMPL
```

in which `stub.nl` is an AMPL generic output file (possibly written by `'ampl -obstub'` or `'ampl -ogstub'`). After solving the problem, `coptampl` writes a `stub.sol` file for use by AMPL's solve and solution commands. When you run AMPL, this all happens automatically if you give the AMPL commands:

```
ampl: option solver coptampl;
ampl: solve;
```

6.6.1 Installation Guide

To use `coptampl` in AMPL, you must have a valid AMPL license and make sure that you have installed Cardinal Optimizer and setup its license properly, see *How to install Cardinal Optimizer* for details. Be sure to check if it satisfies the following requirements for different operating systems.

Windows

On Windows platform, the `coptampl.exe` utility and the `copt.dll` dynamic library contained in the Cardinal Optimizer must appear somewhere in your user or system `PATH` environment variable (or in the current directory).

To test if your setting meets the above requirements, you can check it by executing commands below in command prompt:

```
coptampl -v
```

And you are expected to see output similar to the following on screen:

```
AMPL/x-COPT Optimizer [6.5.2] (windows-x86), driver(20220526), MP(20220526)
```

If the commands failed, then you should recheck your settings.

Linux

On Linux platform, the `coptampl` utility must appears somewhere in your `$PATH` environment variable, while the `libcopt.so` shared library must appears somewhere in your `$LD_LIBRARY_PATH` environment variable.

Similarly, to test if your setting meets the above requirements, just execute commands below in shell:

```
coptampl -v
```

And you are expected to see output similar to the following on screen:

```
AMPL/x-COPT Optimizer [6.5.2] (linux-x86), driver(20220526), MP(20220526)
```

If the commands failed, please recheck your settings.

MacOS

On MacOS platform, the `coptampl` utility must appears somewhere in your `$PATH` environment variable, while the `libcopt.dylib` dynamic library must appears somewhere in your `$DYLD_LIBRARY_PATH` environment variable.

You can execute commands below in shell to see if your settings meets the above requirements:

```
coptampl -v
```

And you are expected to see output similar to the following on screen:

```
AMPL/x-COPT Optimizer [6.5.2] (macos-x86), driver(20220526), MP(20220526)
```

If the commands failed, then please recheck your settings.

6.6.2 Solver Options and Exit Codes

The `coptampl` utility offers some options to customize its behavior. Uers can control it by setting the environment variable `copt_options` or use AMPL's `option` command. To see all available options, please invoke:

```
coptampl -=
```

The supported parameters and their interpretation for current version are shown in Table 6.1:

Table 6.1: Parameters of `coptampl`

Parameter	Interpretation
barhomogeneous	whether to use homogeneous self-dual form in barrier
bariterlimit	iteration limit of barrier method
barthreads	number of threads used by barrier
basis	whether to use or return basis status
bestbound	whether to return best bound by suffix
conflictanalysis	whether to perform conflict analysis
crossoverthreads	number of threads used by crossover
cutlevel	level of cutting-planes generation
divingheurlevel	level of diving heuristics
dualize	whether to dualize a problem before solving it
dualperturb	whether to allow the objective function perturbation
dualprice	specifies the dual simplex pricing algorithm
dualtol	the tolerance for dual solutions and reduced cost
feastol	the feasibility tolerance

continues on next page

Table 6.1 – continued from previous page

heurlevel	level of heuristics
iisfind	whether to compute IIS and return result
iismethod	specify the IIS method
inttol	the integrality tolerance for variables
logging	whether to print solving logs
logfile	name of log file
exportfile	name of model file to be exported
lpmethod	method to solve the LP problem
matrixtol	input matrix coefficient tolerance
mipstart	whether to use initial values for MIP problem
miptasks	number of MIP tasks in parallel
nodecutrounds	rounds of cutting-planes generation of tree node
nodelimit	node limit of the optimization
objno	objective number to solve
count	whether to count the number of solutions
stub	name prefix for alternative MIP solutions written
presolve	level of presolving before solving a problem
relgap	the relative gap of optimization
absgap	the absolute gap of optimization
return_mipgap	whether to return absolute/relative gap by suffix
rootcutlevel	level of cutting-planes generation of root node
rootcutrounds	rounds of cutting-planes generation of root node
roundingheurlevel	level of rounding heuristics
scaling	whether to perform scaling before solving a problem
simplexthreads	number of threads used by dual simplex
sos	whether to use '.sosno' and '.ref' suffix
sos2	whether to use SOS2 to represent piecewise linear terms
strongbranching	level of strong branching
submipheurlevel	level of Sub-MIP heuristics
threads	number of threads to use
timelimit	time limit of the optimization
treecutlevel	level of cutting-planes generation of search tree
wantsol	whether to generate '.sol' file

Please refer to *COPT Parameters* for details.

AMPL uses suffix to store or pass model and solution information, and also some extension features, such as support for SOS constraints. Currently, `coptampl` support suffix information as shown in *Suffix supported by coptampl* :

Table 6.2: Suffix supported by `coptampl`

Suffix	**Interpretation**
absmipgap	absolute gap for MIP problem
bestbound	best bound for MIP problem
iis	store IIS status of variables or constraints
nsol	number of pool solutions written
ref	weight of variable in SOS constraint
relmipgap	relative gap for MIP problem
sos	store type of SOS constraint
sosno	type of SOS constraint
sosref	store variable weight in SOS constraint
sstatus	basis status of variables and constraints

Users who want to know how to use SOS constraints in AMPL, please refer to resources in AMPL's website: How to use SOS constraints in AMPL .

When solving finished, `coptampl` will display a status message and return exit code to AMPL. The exit code can be displayed by:

```
ampl: display solve_result_num;
```

If no solution was found or something unexpected happened, `coptampl` will return non-zero code to AMPL from Table 6.3:

Table 6.3: Exit codes of `coptampl`

Exit Code	Interpretation
0	optimal solution
200	infeasible
300	unbounded
301	infeasible or unbounded
600	user interrupted

6.6.3 Example Usage

The following section will illustrate the use of AMPL by a well-known example called "Diet problem", which finds a mix of foods that satisfies requirements on the amounts of various vitamins, see AMPL book for details.

Suppose the following kinds of foods are available for the following prices per unit, see Table 6.4:

Table 6.4: Prices of foods

Food	Price
BEEF	3.19
CHK	2.59
FISH	2.29
HAM	2.89
MCH	1.89
MTL	1.99
SPG	1.99
TUR	2.49

These foods provide the following percentages, per unit, of the minimum daily requirements for vitamins A, C, B1 and B2, see Table 6.5:

Table 6.5: Nutrition of foods (%)

	A	C	B1	B2
BEEF	60%	20%	10%	15%
CHK	8	0	20	20
FISH	8	10	15	10
HAM	40	40	35	10
MCH	15	35	15	15
MTL	70	30	15	15
SPG	25	50	25	15
TUR	60	20	15	10

The problem is to find the cheapest combination that meets a week's requirements, that is, at least 700% of the daily requirements for each nutrient.

To summarize, the mathematical form for the above problem can be modeled as shown in Eq. 6.6:

$$\begin{aligned} &\text{Minimize:} \\ &\quad \sum_{j \in J} cost_j \cdot buy_j \\ &\text{Subject to:} \\ &\quad n\_min_i \leq \sum_{j \in J} amt_{i,j} \cdot buy_j \leq n\_max_i \ \ \forall i \in I \\ &\quad f\_min_j \leq buy_j \leq f\_max_j \ \ \forall j \in J \end{aligned} \tag{6.6}$$

The AMPL model for above problem is shown in `diet.mod`, see Listing 6.6:

Listing 6.6: `diet.mod`

```
# The code is adopted from:
#
# https://github.com/Pyomo/pyomo/blob/master/examples/pyomo/amplbook2/diet.mod
#
# with some modification by developer of the Cardinal Optimizer

set NUTR;
set FOOD;

param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

param n_min {NUTR} >= 0;
param n_max {i in NUTR} >= n_min[i];

param amt {NUTR, FOOD} >= 0;

var Buy {j in FOOD} >= f_min[j], <= f_max[j];

minimize Total_Cost:
  sum {j in FOOD} cost[j] * Buy[j];

subject to Diet {i in NUTR}:
  n_min[i] <= sum {j in FOOD} amt[i, j] * Buy[j] <= n_max[i];
```

The data file for above problem is shown in `diet.dat`, see Listing 6.7:

Listing 6.7: `diet.dat`

```
# The data is adopted from:
#
# https://github.com/Pyomo/pyomo/blob/master/examples/pyomo/amplbook2/diet.dat
#
# with some modification by developer of the Cardinal Optimizer

data;

set NUTR := A B1 B2 C ;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR ;

param:   cost  f_min  f_max :=
  BEEF   3.19    0     100
  CHK    2.59    0     100
  FISH   2.29    0     100
  HAM    2.89    0     100
  MCH    1.89    0     100
```

(continues on next page)

(continued from previous page)

```
  MTL    1.99    0     100
  SPG    1.99    0     100
  TUR    2.49    0     100 ;

param:   n_min  n_max :=
   A      700   10000
   C      700   10000
   B1     700   10000
   B2     700   10000 ;

param amt (tr):
           A    C   B1   B2 :=
   BEEF   60   20   10   15
   CHK     8    0   20   20
   FISH    8   10   15   10
   HAM    40   40   35   10
   MCH    15   35   15   15
   MTL    70   30   15   15
   SPG    25   50   25   15
   TUR    60   20   15   10 ;
```

To solve the problem with `coptampl` in AMPL, just type commands in command prompt on Windows or shell on Linux and MacOS:

```
ampl: model diet.mod;
ampl: data diet.dat;
ampl: option solver coptampl;
ampl: option copt_options 'logging 1';
ampl: solve;
```

`coptampl` solve it quickly and display solving log and status message on screen:

```
x-COPT 5.0.1: optimal solution; objective 88.2
1 simplex iterations
```

So `coptampl` claimed it found the optimal solution, and the minimal cost is 88.2 units. You can further check the solution by:

```
ampl: display Buy;
```

And you will get:

```
Buy [*] :=
BEEF   0
 CHK   0
FISH   0
 HAM   0
 MCH  46.6667
 MTL   0
 SPG   0
 TUR   0
;
```

So if we buy 46.667 units of MCH, we will have a minimal cost of 88.2 units.

6.7 Pyomo Interface

Pyomo is a Python based, open source optimization modeling language with a diverse set of optimization capabilities. It is used by researchers to solve complex real-world applications, see Who uses Pyomo? for more introduction. The following documentation explains how to use the **Cardinal Optimizer**.

6.7.1 Installation Guide

To use the Cardinal Optimizer in Pyomo, you should setup Pyomo and Cardinal Optimizer correctly first. Pyomo currently supports Python 2.7, 3.6-3.9, you can install Python from Anaconda Distribution of Python or from Official Python. We recommend install Python from Anaconda since it is much more friendly and convenient for fresh users.

Using conda

The recommended way to install Pyomo in Anaconda Distribution of Python is to use `conda` which is built-in supported. Just execute the following commands in command prompt on Windows or shell on Linux and MacOS:

```
conda install -c conda-forge pyomo
```

Pyomo also has conditional dependencies on a variety of third-party Python packages, they can be installed using `conda` with commands:

```
conda install -c conda-forge pyomo.extras
```

Using pip

The alternative way to install Pyomo is to use the standard `pip` utitility, just execute the following commands in command prompt on Windows or shell on Linux and MacOS:

```
pip install pyomo
```

If you encounter any problems when installing Pyomo, please refer to How to install Pyomo for details. To install Cardinal Optimizer and setup its license properly, please refer to *How to install Cardinal Optimizer* for details.

6.7.2 Example Usage

We are going to make a simple introduction on how to use the Cardinal Optimizer in Pyomo by solving the example described in *AMPL Interface - Example Usage*. Users who want to learn more information about Pyomo may refer to Pyomo documentation for details.

Abstract Model

Pyomo provides two major approaches to construct any supported model types, here we show the **Abstract Model** approach to solve the above problem.

The source code `pydiet_abstract.py` is shown below, see Listing 6.8:

Listing 6.8: pydiet_abstract.py

```
# The code is adopted from:
#
# https://github.com/Pyomo/pyomo/blob/master/examples/pyomo/amplbook2/diet.py
#
# with some modification by developer of the Cardinal Optimizer

from pyomo.core import *

model = AbstractModel()

model.NUTR = Set()
model.FOOD = Set()

model.cost  = Param(model.FOOD, within=NonNegativeReals)
model.f_min = Param(model.FOOD, within=NonNegativeReals)

model.f_max = Param(model.FOOD)
model.n_min = Param(model.NUTR, within=NonNegativeReals)
model.n_max = Param(model.NUTR)
model.amt   = Param(model.NUTR, model.FOOD, within=NonNegativeReals)

def Buy_bounds(model, i):
  return (model.f_min[i], model.f_max[i])
model.Buy = Var(model.FOOD, bounds=Buy_bounds)

def Objective_rule(model):
  return sum_product(model.cost, model.Buy)
model.totalcost = Objective(rule=Objective_rule, sense=minimize)

def Diet_rule(model, i):
  expr = 0

  for j in model.FOOD:
    expr = expr + model.amt[i, j] * model.Buy[j]

  return (model.n_min[i], expr, model.n_max[i])
model.Diet = Constraint(model.NUTR, rule=Diet_rule)
```

And the data file pydiet_abstract.dat in Listing 6.9:

Listing 6.9: pydiet_abstract.dat

```
# The data is adopted from:
#
# https://github.com/Pyomo/pyomo/blob/master/examples/pyomo/amplbook2/diet.dat
#
# with some modification by developer of the Cardinal Optimizer

data;

set NUTR := A B1 B2 C ;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR ;

param:   cost  f_min  f_max :=
  BEEF   3.19    0     100
  CHK    2.59    0     100
  FISH   2.29    0     100
  HAM    2.89    0     100
  MCH    1.89    0     100
  MTL    1.99    0     100
```

(continues on next page)

(continued from previous page)

```
  SPG    1.99    0     100
  TUR    2.49    0     100 ;

param:   n_min  n_max :=
   A      700   10000
   C      700   10000
   B1     700   10000
   B2     700   10000 ;

param amt (tr):
           A    C   B1   B2 :=
   BEEF   60   20   10   15
   CHK     8    0   20   20
   FISH    8   10   15   10
   HAM    40   40   35   10
   MCH    15   35   15   15
   MTL    70   30   15   15
   SPG    25   50   25   15
   TUR    60   20   15   10 ;
```

To solve the problem using Pyomo and the Cardinal Optimizer, just type commands below in command prompt on Windows or Bash shell on Linux and MacOS.

```
pyomo solve --solver=coptampl pydiet_abstract.py pydiet_abstract.dat
```

When solving the problem, Pyomo write log information to the screen:

```
[    0.00] Setting up Pyomo environment
[    0.00] Applying Pyomo preprocessing actions
[    0.00] Creating model
[    0.01] Applying solver
[    0.05] Processing results
    Number of solutions: 1
    Solution Information
      Gap: None
      Status: optimal
      Function Value: 88.19999999999999
    Solver results file: results.yml
[    0.05] Applying Pyomo postprocessing actions
[    0.05] Pyomo Finished
```

Upon completion, you can check the solution summary in `results.yml`:

```
# ==========================================================
# = Solver Results                                         =
# ==========================================================
# ----------------------------------------------------------
#   Problem Information
# ----------------------------------------------------------
Problem:
- Lower bound: -inf
  Upper bound: inf
  Number of objectives: 1
  Number of constraints: 4
  Number of variables: 8
  Sense: unknown
# ----------------------------------------------------------
#   Solver Information
# ----------------------------------------------------------
Solver:
- Status: ok
```

(continues on next page)

(continued from previous page)

```
  Message: COPT-AMPL\x3a optimal solution; objective 88.2, iterations 1
  Termination condition: optimal
  Id: 0
  Error rc: 0
  Time: 0.03171110153198242
# ------------------------------------------------------------
#   Solution Information
# ------------------------------------------------------------
Solution:
- number of solutions: 1
  number of solutions displayed: 1
- Gap: None
  Status: optimal
  Message: COPT-AMPL\x3a optimal solution; objective 88.2, iterations 1
  Objective:
    totalcost:
      Value: 88.19999999999999
  Variable:
    Buy[MCH]:
      Value: 46.666666666666664
  Constraint: No values
```

So the minimal total cost is about 88.2 units when buying 46.67 units of MCH.

Concrete Model

The other approach to construct model in Pyomo is to use **Concrete Model**, we will show how to model and solve the above problem in this way.

Concrete models can be solved using the `"Direct"` and `"Persistent"` interface methods. This method relies on the Pyomo plugin file `"copt_pyomo.py"` of COPT, which is located in the `"lib/pyomo"` subfolder of the installation package.

To use this plugin, you need to copy the `"copt_pyomo.py"` file to the same directory of your program, and have correctly installed the corresponding version of the Python interface of COPT(`coptpy`).

The source code `pydiet_concrete.py` is shown in Listing 6.10:

Listing 6.10: `pydiet_concrete.py`

```
# The code is adopted from:
#
# https://github.com/Pyomo/pyomo/blob/master/examples/pyomo/amplbook2/diet.py
#
# with some modification by developer of the Cardinal Optimizer

from __future__ import print_function, division

import pyomo.environ as pyo
import pyomo.opt as pyopt

from copt_pyomo import *

# Nutrition set
NUTR = ["A", "C", "B1", "B2"]
# Food set
FOOD = ["BEEF", "CHK", "FISH", "HAM", "MCH", "MTL", "SPG", "TUR"]

# Price of foods
cost = {"BEEF": 3.19, "CHK": 2.59, "FISH": 2.29, "HAM": 2.89, "MCH": 1.89,
        "MTL":  1.99, "SPG": 1.99, "TUR":  2.49}
```

(continues on next page)

(continued from previous page)

```
# Nutrition of foods
amt = {"BEEF": {"A": 60, "C": 20, "B1": 10, "B2": 15},
       "CHK":  {"A": 8,  "C": 0,  "B1": 20, "B2": 20},
       "FISH": {"A": 8,  "C": 10, "B1": 15, "B2": 10},
       "HAM":  {"A": 40, "C": 40, "B1": 35, "B2": 10},
       "MCH":  {"A": 15, "C": 35, "B1": 15, "B2": 15},
       "MTL":  {"A": 70, "C": 30, "B1": 15, "B2": 15},
       "SPG":  {"A": 25, "C": 50, "B1": 25, "B2": 15},
       "TUR":  {"A": 60, "C": 20, "B1": 15, "B2": 10}}

# The "diet problem" using ConcreteModel
model = pyo.ConcreteModel()

model.NUTR = pyo.Set(initialize=NUTR)
model.FOOD = pyo.Set(initialize=FOOD)

model.cost = pyo.Param(model.FOOD, initialize=cost)

def amt_rule(model, i, j):
  return amt[i][j]
model.amt  = pyo.Param(model.FOOD, model.NUTR, initialize=amt_rule)

model.f_min = pyo.Param(model.FOOD, default=0)
model.f_max = pyo.Param(model.FOOD, default=100)

model.n_min = pyo.Param(model.NUTR, default=700)
model.n_max = pyo.Param(model.NUTR, default=10000)

def Buy_bounds(model, i):
  return (model.f_min[i], model.f_max[i])
model.buy = pyo.Var(model.FOOD, bounds=Buy_bounds)

def Objective_rule(model):
  return pyo.sum_product(model.cost, model.buy)
model.totalcost = pyo.Objective(rule=Objective_rule, sense=pyo.minimize)

def Diet_rule(model, j):
  expr = 0

  for i in model.FOOD:
    expr = expr + model.amt[i, j] * model.buy[i]

  return (model.n_min[j], expr, model.n_max[j])
model.Diet = pyo.Constraint(model.NUTR, rule=Diet_rule)

# Reduced costs of variables
model.rc = pyo.Suffix(direction=pyo.Suffix.IMPORT)

# Activities and duals of constraints
model.slack = pyo.Suffix(direction=pyo.Suffix.IMPORT)
model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)

# Use 'copt_direct' solver to solve the problem
solver = pyopt.SolverFactory('copt_direct')

# Use 'copt_persistent' solver to solve the problem
# solver = pyopt.SolverFactory('copt_persistent')
# solver.set_instance(model)

results = solver.solve(model, tee=True)

```

(continues on next page)

(continued from previous page)

```
# Check result
print("")
if results.solver.status == pyopt.SolverStatus.ok and \
   results.solver.termination_condition == pyopt.TerminationCondition.optimal:
  print("Optimal solution found")
else:
  print("Something unexpected happened: ", str(results.solver))

print("")
print("Optimal objective value:")
print("  totalcost: {0:6f}".format(pyo.value(model.totalcost)))

print("")
print("Variables solution:")
for i in FOOD:
  print("  buy[{0:4s}] = {1:9.6f} (rc: {2:9.6f})".format(i, \
                                                  pyo.value(model.buy[i]), \
                                                  model.rc[model.buy[i]]))

print("")
print("Constraint solution:")
for i in NUTR:
  print("  diet[{0:2s}] = {1:12.6f} (dual: {2:9.6f})".format(i, \
                                                      model.slack[model.Diet[i]], \
                                                      model.dual[model.Diet[i]]))
```

To solve the problem using Pyomo and the Cardinal Optimizer, just execute commands below:

```
python pydiet_concrete.py
```

Up completion, you should see solution summary on screen as below:

```
Optimal solution found
Objective:
  totalcost: 88.200000
Variables:
  buy[BEEF] = 0.000000
  buy[CHK ] = 0.000000
  buy[FISH] = 0.000000
  buy[HAM ] = 0.000000
  buy[MCH ] = 46.666667
  buy[MTL ] = 0.000000
  buy[SPG ] = 0.000000
  buy[TUR ] = 0.000000
```

So the Cardinal Optimizer found the optimal solution, which is about 88.2 units when buying about 46.67 units of MCH.

6.8 PuLP Interface

PuLP is an open source modeling tool based on Python, it is mainly used for modeling integer programming problems. This chapter introduces how to use the Cardinal Optimizer (COPT) in PuLP.

6.8.1 Installation guide

Before calling COPT in PuLP to solve problem, users need to setup PuLP and COPT correctly. PuLP currently supports Python 2.7 and later versions of Python. Users can download Python from Anaconda distribution or Python official distribution . We recommend users install the Anaconda distribution, because it is more user-friendly and convenient for Python novices.

Install via conda

We recommend that users who have installed the Anaconda distribution of Python use its own `conda` tool to install PuLP. Execute the following commands in Windows command prompt or terminal on Linux and MacOS:

```
conda install -c conda-forge pulp
```

Install via pip

Users can also install PuLP through the standard `pip` tool, execute the following command in Windows command prompt or Linux and MacOS terminal:

```
pip install pulp
```

6.8.2 Setup PuLP interface

The PuLP interface of COPT is the Python file `copt_pulp.py` in the COPT package. After installing and configuring COPT, users only need to put the interface file in the project directory and import the plug-in:

```
from copt_pulp import *
```

In the solving function `solve`, specify the solver to COPT to solve:

```
prob.solve(COPT_DLL())
```

or:

```
prob.solve(COPT_CMD())
```

6.8.3 Introduction of features

The PuLP interface of COPT provides two methods: command line and dynamic library, which are introduced as follows:

Command line

The command-line method actually calls the interactive tool `copt_cmd` of COPT to solve problems. In this way, PuLP generates the MPS format file corresponding to the model, and combines the parameter settings passed by the user to generate the solving commands. Upon finish of solving, COPT writes and reads the result file, and assigns values to the corresponding variables and return them to PuLP.

Functions of the command line method are encapsulated as class `COPT_CMD`. Users can set parameters when creating the object of the class and provides the following parameters:

- `keepFiles`

  This option controls whether to keep the generated temporary files. The default value is `0`, which means no temporary files are kept.

- `mip`

  This option controls whether to support solving integer programming models. The default value is `True`, which means support solving integer programming models.

- `msg`

  This option controls whether to print log information on the screen. The default value is `True`, that is, print log information.

- `mip_start`

  This option controls whether to use initial solution information for integer programming models. The default value is `False`, that is, the initial solution information will not be used.

- `logfile`

  This option specifies the solver log. The default value is `None`, which means no solver log will be generated.

- `params`

  This option sets optimization parameters in the form of `key=value`. Please refer to the chapter *Parameters* for currently supported parameters.

Dynamic library

The dynamic library method directly calls COPT C APIs to solve problems. In this way, PuLP generates problem data and call COPT APIs to load the problem and parameters set by the user. When optimization finishes, the solution is obtained by calling COPT APIs, and then assigned to the corresponding variables and constraints, and passed back to PuLP.

Functions of the dynamic library method are encapsulated as class `COPT_DLL`. Users can set parameters when creating the object of the class and provides the following parameters:

- `mip`

  This option controls whether to support solving integer programming models. The default value is `True`, which means support solving integer programming models.

- `msg`

  This option controls whether to print log information on the screen. The default value is `True`, that is, print log information.

- `mip_start`

  This option controls whether to use initial solution information for integer programming models. The default value is `False`, that is, the initial solution information will not used.

- `logfile`

This option specifies the solver log. The default value is `None`, which means no solver log will be generated.

- `params`

  This option sets optimization parameters in the form of `key=value`. Please refer to the chapter *Parameters* for currently supported parameters.

In addition, the following methods are provided:

- `setParam(self, name, val)`

  Set optimization solution parameters.

- `getParam(self, name)`

  Obtain optimized solution parameters.

- `getAttr(self, name)`

  Get the attribute information of the model.

- `write(self, filename)`

  Output MPS/LP format model file, COPT binary format model file, result file, basic solution file, initial solution file and parameter setting file.

6.9 CVXPY Interface

CVXPY is an open source Python-embedded modeling language for convex optimization problems. It lets you express your problem in a natural way that follows the math, which is quite flexible and efficient. This chapter introduces how to use the Cardinal Optimizer (COPT) in CVXPY.

6.9.1 Installation guide

Before calling COPT in CVXPY to solve problem, users need to setup CVXPY and COPT correctly. CVXPY currently supports Python 3.6 and later versions of Python. Users can download Python from Anaconda distribution or Python official distribution . We recommend users install the Anaconda distribution, because it is more user-friendly and convenient for Python novices.

Install via conda

We recommend that users who have installed the Anaconda distribution of Python use its own `conda` tool to install CVXPY. Execute the following commands in Windows command prompt or terminal on Linux and MacOS:

```
conda install -c conda-forge cvxpy
```

Install via pip

Users can also install CVXPY through the standard `pip` tool, execute the following command in Windows command prompt or Linux and MacOS terminal:

```
pip install cvxpy
```

6.9.2 Setup CVXPY interface

The CVXPY interface of COPT is the Python file `copt_cvxpy.py` in the COPT package, and also depends on the Python interface of COPT. Suppose Python version is 3.7, users should install the Python 3.7 version of COPT Python interface, see *Python Interface* for detailed install guide. After installing and configuring COPT, users only need to put the interface file in the project directory and import the plugin:

```
from copt_cvxpy import *
```

In the solving function `solve`, specify the solver to COPT to solve:

```
prob.solve(solver=COPT())
```

6.9.3 Introduction of features

The CVXPY interface of COPT supports Linear Programming (LP), Mixed Integer Programming (MIP), Quadratic Programming (QP) and Second-Order-Cone Programming (SOCP), common used parameters are:

- `verbose`

  CVXPY builtin parameter, which controls whether to display solving log to the screen. The default value is `False`, which means no log to be displayed.

- `params`

  This option sets optimization parameters in the form of `key=value`. Please refer to the chapter *Parameters* for currently supported parameters.

Chapter 7

General Constants

There are three types of constants.

1. Constructing models, such as optimization directions, constraint senses or variable types.
2. Accessing solution results, such as API return code, basis status and LP status.
3. Monitoring optimization progress, such as callback context.

7.1 Version information

- `VERSION_MAJOR`

  The major version number.

- `VERSION_MINOR`

  The minor version number.

- `VERSION_TECHNICAL`

  The technical version number.

7.2 Optimization directions

For different optimization scenarios, it may be required to either maximize or minimize the objective function. There are two optimization directions:

- `MINIMIZE`

  For minimizing the objective function.

- `MAXIMIZE`

  For maximizing the objective function.

The optimization direction is automatically set when reading in a problem from file. Besides, COPT provides relevant functions, allowing user to explicitly set. Functions for different APIs are listed below:

Table 7.1: Functions for setting optimization directions

Programming API	Function
C	`COPT_SetObjSense`
C++	`Model::SetObjSense()`
C#	`Model.SetObjSense()`
Java	`Model.setObjSense()`
Python	`Model.setObjSense()`

NOTE: The function names, calling methods and parameter names in different programming interfaces are slightly different. For details, please refer to the API parameters of each programming language.

7.3 Infinity and Undefined Value

Infinity

In COPT, the infinite bound is represented by a large value, whose default value is also available as a constant:

- `INFINITY`

  The default value (`1e30`) of the infinite bound.

Undefined Value

In COPT, the undefined value is represented by another large value. For example, the default solution value of MIP start is set to a constant:

- `UNDEFINED`

  Undefined value(`1e40`).

7.4 Constraint senses

Traditionally, for optimization models, constraints are defined using **senses**. The most common constraint senses are:

- `LESS_EQUAL`

  For constraint in the form of $g(x) \leq b$

- `GREATER_EQUAL`

  For constraint in the form of $g(x) \geq b$

- `EQUAL`

  For constraint in the form of $g(x) = b$

In additional, there are two less used constraint senses:

- `FREE`

  For unconstrained expression

- `RANGE`

  For constraints with both lower and upper bounds in the form of $l \leq g(x) \leq u$.

NOTE: Using constraint senses is supported by COPT but not recommended. We recommend defining constraints using explicit lower and upper bounds.

7.5 Variable types

Variable types are used for defining whether a variable is continuous or integral.

- `CONTINUOUS`

  Non-integer continuous variables

- `BINARY`

  Binary variables

- `INTEGER`

  Integer variables

7.6 SOS-constraint types

SOS constraint (Special Ordered Set) is a kind of special constraint that places restrictions on the values that a set of variables can take. COPT currently support two types of SOS constraints:

- `SOS_TYPE1`

  SOS1 constraint

  At most one variable in the constraint is allowed to take a non-zero value.

- `SOS_TYPE2`

  SOS2 constraint

  At most two variables in the constraint are allowed to take non-zero value, and those non-zero variables must be contiguous.

NOTE: Variables in SOS constraints are allowed to be continuous, binary and integer.

7.7 Indicator constraint

Indicator constraint is a kind of logical constraints, is uses a binary variable y as the indicator variable, and implies whether the linear constraint $a^T x \leq b$ is valid based on value of variable y. The canonical form of an indicator constraint is:

$$y = f \rightarrow a^T x \leq b \tag{7.1}$$

Where $f \in \{0, 1\}$. If $y = f$, the linear constraint is valid. Otherwise if $y \neq f$, the linear constraint is invalid (may be violated). The sense of the linear constraint can be $\leq$, $\geq$ and $=$.

7.8 Second-Order-Cone constraint

The Second-Order-Cone (SOC) constraint is a special type of quadratic constraints. Currently, COPT supports two types of SOC constraints:

- `CONE_QUAD` : Regular cone

$$Q^n = \left\{ x \in \mathbb{R}^n \;\middle|\; x_0 \geq \sqrt{\sum_{i=1}^{n-1} x_i^2}, x_0 \geq 0 \right\} \tag{7.2}$$

- `CONE_RQUAD` : Rotated cone

$$Q_r^n = \left\{ x \in \mathbb{R}^n \;\middle|\; 2x_0 x_1 \geq \sum_{i=2}^{n-1} x_i^2, x_0 \geq 0, x_1 \geq 0 \right\} \tag{7.3}$$

7.9 Quadratic objective function

Besides linear objective function, COPT also supports general convex quadratic objective function.

The mathematical form is:

$$x^T Q x + c^T x \tag{7.4}$$

Where, x is an array of variables, Q is the quadratic part of the quadratic objective funtion and c is the linear part.

7.10 Quadratic constraint

Besides the special type of quadratic constraint, Second-Order-Cone (SOC) constraint, COPT also supports general convex quadratic constraint.

The mathematical form is:

$$x^T Q x + q^T x \leq b \tag{7.5}$$

Where, x is an array of variables, Q is the quadratic part of the quadratic constraint and c is the linear part.

7.11 Basis status

For an LP problem with n variables and m constraints, the constraints are treated as slack variables *internally*, resulting in $n + m$ variables. When solving an LP problem using the simplex method, the simplex method fixes n variables at one of their bounds, and then computes solutions for the other m variables. The m variables with computed solution are called *basic* variables, and the other n variables are called *non-basic* variables. The simplex progress and its final solution can be defined using the basis status of all the variables and constraints. The basis status supported by COPT are listed below:

Table 7.2: Basis status values and descriptions

Basis status codes	Value	Description
`BASIS_LOWER`	0	The variable is non-basic at its lower bound
`BASIS_BASIC`	1	The variable is basic
`BASIS_UPPER`	2	The variable is non-basic at its upper bound
`BASIS_SUPERBASIC`	3	The variable is non-basic but not any of its bounds
`BASIS_FIXED`	4	The variable is non-basic and fixed at its bound

7.12 Solution status

The solution status of an problem is called solution status, Possible solution status values are listed below:

Table 7.3: Solution Status

Status Codes	Description
UNSTARTED	The solving process is not started yet
OPTIMAL	Optimal values founded
INFEASIBLE	The model is infeasible
UNBOUNDED	The objective is unbounded
INF_OR_UNB	The model is infeasible or unbounded
NUMERICAL	Numerical trouble encountered
NODELIMIT	Solving process not finished within node limit
TIMEOUT	Solving process not finished within time limit
UNFINISHED	The solving process is stopped but solver cannot provide a solution because of numerical issues
IMPRECISE	The solution is imprecise
INTERRUPTED	The solving process is stopped by user interrupt

Notes

- In the Python API, solution states are defined in COPT's General Constants Class. They can be accessed via the `"COPT."` prefix or `Model.status` ;
- In the Constant class of the Java API and C# API, the constants about the solution status are defined in the `Status` class;
- The linear programming solution status can be obtained through the attribute `"LpStatus"` , and the integer programming solution status can be obtained through the attribute `"MipStatus"` .

7.13 Client configuration

For floating and cluster clients, user are allowed to set client configuration parameters, currently available settings are:

- `CLIENT_CLUSTER`

  IP address of cluster server.

- `CLIENT_FLOATING`

  IP address of token server.

- `CLIENT_PASSWORD`

  Password of cluster server.

- `CLIENT_PORT`

  Connection port of token server.

- `CLIENT_WAITTIME`

  Wait time of client.

7.14 Callback context

- `CBCONTEXT_MIPRELAX`

  Invokes the callback when a new LP-relaxation solution is found.

- `CBCONTEXT_MIPSOL`

  Invokes the callback when a new MIP candidate solution is found.

7.15 Methods for accessing constants

In different programming interfaces, the ways of accessing constants are slightly different. In the C language interface, the constant name is prefixed with `"COPT_"` (like `COPT_MAXIMIZE`). For details, please refer to the corresponding chapters of each programming language API reference manual:

- C API: *C API Reference: Constants*
- C++ API: *C++ API Reference: Constants*
- C# API: *C# API Reference: General Constants*
- Java API: *Java API Reference: General Constants*
- Python API: *Python API Reference: General Constants*

Chapter 8

Attributes

To query and modify properties of a COPT model is through the attribute interface. A variety of different attributes are available, and they can be associated with solutions, or the model.

8.1 Problem related

Problem related attributes provide the relevant information of the model composition and description. The names and descriptions of these attributes are summarized below.

Table 8.1: Problem related attributes

Name	Type	Description
Cols	Integer	Number of variables (columns) in the problem
PSDCols	Integer	Number of PSD variables in the problem
Rows	Integer	Number of constraints (rows) in the problem
Elems	Integer	Number of non-zero elements in the coefficient matrix
QElems	Integer	Number of non-zero quadratic elements in the quadratic objective function
PSDElems	Integer	Number of PSD terms in objective function
SymMats	Integer	Number of symmetric matrices in the problem
Bins	Integer	Number of binary variables
Ints	Integer	Number of integer variables
Soss	Integer	Number of SOS constraints
Cones	Integer	Number of Second-Order-Cone constraints
QConstrs	Integer	Number of quadratic constraints
PSDConstrs	Integer	Number of PSD constraints
Indicators	Integer	Number of indicator constraints
ObjSense	Integer	The optimization direction
ObjConst	Double	The constant part of the objective function
HasQObj	Integer	Whether the problem has quadratic objective function
HasPSDObj	Integer	Whether the problem has PSD terms in objective function
IsMIP	Integer	Whether the problem is a MIP

- `Cols`

  Integer attribute.

  Number of variables (columns) in the problem.

- `PSDCols`

  Integer attribute.

Number of PSD variables in the problem.

- `Rows`

  Integer attribute.

  Number of constraints (rows) in the problem.

- `Elems`

  Integer attribute.

  Number of non-zero elements in the coefficient matrix.

- `QElems`

  Integer attribute.

  Number of non-zero quadratic elements in the quadratic objective function.

- `PSDElems`

  Integer attribute.

  Number of PSD terms in objective function.

- `SymMats`

  Integer attribute.

  Number of symmetric matrices in the problem.

- `Bins`

  Integer attribute.

  Number of binary variables.

- `Ints`

  Integer attribute.

  Number of integer variables.

- `Soss`

  Integer attribute.

  Number of SOS constraints.

- `Cones`

  Integer attribute.

  Number of Second-Order-Cone constraints.

- `QConstrs`

  Integer attribute.

  Number of quadratic constraints.

- `PSDConstrs`

  Integer attribute.

  Number of PSD constraints.

- `Indicators`

  Integer attribute.

  Number of indicator constraints.

- `ObjSense`

Integer attribute.

The optimization direction.

- `ObjConst`

  Double attribute.

  The constant part of the objective function.

- `HasQObj`

  Integer attribute.

  Whether the problem has quadratic objective function.

- `HasPSDObj`

  Integer attribute.

  Whether the problem has PSD terms in objective function.

- `IsMIP`

  Integer attribute.

  Whether the problem is a MIP.

8.2 Solution related

Solution related attributes provide the relevant information of the solution composition and description. The names and descriptions of these attributes are summarized below.

Table 8.2: Solution related attributes

Name	Type	Description
LpStatus	Integer	The LP status
MipStatus	Integer	The MIP status
SimplexIter	Integer	Number of simplex iterations performed
BarrierIter	Integer	Number of barrier iterations performed
NodeCnt	Integer	Number of explored nodes
PoolSols	Integer	Number of solutions in solution pool
TuneResults	Integer	Number of parameter tuning results
HasLpSol	Integer	Whether LP solution is available
HasBasis	Integer	Whether LP basis is available
HasDualFarkas	Integer	Whether the dual Farkas of an infeasible LP problem is available
HasPrimalRay	Integer	Whether the primal ray of an unbounded LP problem is available
HasMipSol	Integer	Whether MIP solution is available
IISCols	Integer	Number of bounds of columns in IIS
IISRows	Integer	Number of rows in IIS
IISSOSs	Integer	Number of SOS constraints in IIS
IISIndicators	Integer	Number of indicator constraints in IIS
HasIIS	Double	Whether IIS is available
HasFeasRelaxSol	Integer	Whether feasibility LP-relaxation solution is available
IsMinIIS	Integer	Whether the computed IIS is minimal
LpObjval	Integer	The LP objective value
BestObj	Double	Best integer objective value for MIP
BestBnd	Double	Best bound for MIP
BestGap	Double	Best relative gap for MIP
FeasRelaxObj	Double	Feasibility relaxation objective value
SolvingTime	Double	The time spent for the optimization (in seconds)

- `LpStatus`

  Integer attribute.

  The LP status. Please refer to *General Constants: Solution Status* for possible values.

- `MipStatus`

  Integer attribute.

  The MIP status. Please refer to *General Constants: Solution Status* for possible values.

- `SimplexIter`

  Integer attribute.

  Number of simplex iterations performed.

- `BarrierIter`

  Integer attribute.

  Number of barrier iterations performed.

- `NodeCnt`

  Integer attribute.

  Number of explored nodes.

- `PoolSols`

  Integer attribute.

  Number of solutions in solution pool.

- `TuneResults`

  Integer attribute.

  Number of parameter tuning results

- `HasLpSol`

  Integer attribute.

  Whether LP solution is available.

- `HasBasis`

  Integer attribute.

  Whether LP basis is available.

- `HasDualFarkas`

  Integer attribute.

  Whether the dual Farkas of an infeasible LP problem is available.

- `HasPrimalRay`

  Integer attribute.

  Whether the primal ray of an unbounded LP problem is available.

- `HasMipSol`

  Integer attribute.

  Whether MIP solution is available.

- `IISCols`

  Integer attribute.

  Number of bounds of columns in IIS.

- `IISRows`

  Integer attribute.

  Number of rows in IIS.

- `IISSOSs`

  Integer attribute.

  Number of SOS constraints in IIS.

- `IISIndicators`

  Integer attribute.

  Number of indicator constraints in IIS.

- `HasIIS`

  Integer attribute.

  Whether IIS is available.

- `HasFeasRelaxSol`

  Integer attribute.

  Whether feasibility LP-relaxation solution is available.

- `IsMinIIS`

  Integer attribute.

  Whether the computed IIS is minimal.

- `LpObjval`

  Double attribute.

  The LP objective value.

- `BestObj`

  Double attribute.

  Best integer objective value for MIP.

- `BestBnd`

  Double attribute.

  Best bound for MIP.

- `BestGap`

  Double attribute.

  Best relative gap for MIP.

- `FeasRelaxObj`

  Double attribute.

  Feasibility relaxation objective value.

- `SolvingTime`

  Double attribute.

  The time spent for the optimization (in seconds).

8.3 Methods for accessing attributes

In different programming interfaces, the ways to access attributes are slightly different. For details, please refer to the corresponding chapters of each programming language API reference manual:

- C API: *C API Functions: Accessing attributes*
- C++ API: *C++ API Reference: Attributes*
- C# API: *C# API Reference: Attributes*
- Java API: *Java API Reference: Attributes*
- Python API: *Python API Reference: Attributes*

Chapter 9

Information

The information constants describe the relevant information of the model components (objective function, constraints and variables), solution results, and feasibility relaxation calculation results. This chapter will introduce the information constants provided by COPT and their meanings.

Table 9.1: COPT information

Name	Type	Description
Obj	Double	Objective cost of columns
LB	Double	Lower bounds of columns or rows
UB	Double	Upper bounds of columns or rows
Value	Double	Solution of columns
Slack	Double	Solution of slack variables, also known as activities of constraints. Only available for LP problem
Dual	Double	Solution of dual variables. Only available for LP problem
RedCost	Double	Reduced cost of columns. Only available for LP problem
DualFarkas	Double	The dual Farkas for constraints of an infeasible LP problem
PrimalRay	Double	The primal ray for variables of an unbounded LP problem
RelaxLB	Double	Feasibility relaxation values for lower bounds of columns or rows
RelaxUB	Double	Feasibility relaxation values for upper bounds of columns or rows
BestObj	Double	Current best objective
BestBnd	Double	Current best objective bound
HasIncumbent	Integer	Whether an incumbent is available
MipCandObj	Double	Objective value for current feasible solution candidate
RelaxSolObj	Double	Current Objective of LP-relaxation

9.1 Problem information

- `Obj`

  Double information.

  Objective cost of columns.

- `LB`

  Double information.

  Lower bounds of columns or rows.

- `UB`

  Double information.

  Upper bounds of columns or rows.

9.2 Solution information

- `Value`

  Double information.

  Solution of columns.

- `Slack`

  Double information.

  Solution of slack variables, also known as activities of constraints. Only available for LP problem.

- `Dual`

  Double information.

  Solution of dual variables. Only available for LP problem.

- `RedCost`

  Double information.

  Reduced cost of columns. Only available for LP problem.

9.3 Dual Farkas and primal ray

Advanced topic. When an LP is infeasible or unbounded, the solver can return the dual Farkas or primal ray to prove it.

- `DualFarkas`

  Double information.

  The dual Farkas for constraints of an infeasible LP problem. Please enable the parameter `"ReqFarkasRay"` to ensure that the dual Farkas is available when the LP is infeasible.

  Without loss of generality, the concept of the dual Farkas can be conveniently demonstrated using an LP problem with general variable bounds and equality constraints: $Ax = 0$ and $l \leq x \leq u$. When the LP is infeasible, a dual Farkas vector y can prove that the system has conflict that $\max y^T Ax < y^T b = 0$. Computing $\max y^T Ax$: with the vector $\hat{a} = y^T A$, choosing variable bound $x_i = l_i$ when $\hat{a}_i < 0$ and $x_i = u_i$ when $\hat{a}_i > 0$ gives the maximal possible value of $y^T Ax$ for any x within their bounds.

  Some application relies on the alternate conflict $\min \bar{y}^T Ax > \bar{y}^T b = 0$. This can be achieved by negating the dual Farkas, i.e. $\bar{y} = -y$ returned by the solver.

  In very rare cases, the solver may fail to return a valid dual Farkas. For example when the LP problem slightly infeasible by tiny amount, which We recommend to study and to repair the infeasibility using FeasRelax instead.

- `PrimalRay`

  Double information.

  The primal ray for variables of an unbounded LP problem. Please enable the parameter `"ReqFarkasRay"` to ensure that the primal ray is available when an LP is unbounded.

  For a minimization LP problem in the standard form: $\min c^T x, Ax = b$ and $x \geq 0$, a primal ray vector r satisfies that $r \geq 0, Ar = 0$ and $c^T r < 0$.

9.4 Feasibility relaxation information

- `RelaxLB`

  Double information.

  Feasibility relaxation values for lower bounds of columns or rows.

- `RelaxUB`

  Double information.

  Feasibility relaxation values for upper bounds of columns or rows.

9.5 Callback information

- `BestObj`

  Double information.

  Current best objective.

- `BestBnd`

  Double information.

  Current best objective bound.

- `HasIncumbent`

  Integer information.

  Whether an incumbent is available.

- `MipCandObj`

  Double information.

  Objective value for current feasible solution candidate.

- `RelaxSolObj`

  Double information.

  Current Objective of LP-relaxation.

9.6 Methods for accessing information

In different programming interfaces, the ways to access and set information are slightly different. For details, please refer to the corresponding chapters of each programming language API reference manual:

- C API: *C API Functions: Accessing information of problem*
- C# API: *C# API Reference: Information*
- Java API: *Java API Reference: Information*
- Python API: *Python API Reference: Information*

Chapter 10

Parameters

Parameters control the operation of the **Cardinal Optimizer**. They can be modified before the optimization begins. Each parameter has its own default value and value range. Before starting the solution, user can set the parameters to different values, so as to put forward specific requirements on the solution algorithm and solution process. Of course, the default settings can also be maintained.

According to the task performed by the solver COPT and the optimization problem solved, it can be divided into different types of parameters.

This chapter will introduce the parameters constants provided by COPT and their meanings.

10.1 Limits and tolerances

Table 10.1: Limits and tolerances parameters

Name	Type	Description
TimeLimit	Double	Time limit of the optimization
NodeLimit	Integer	Node limit of the optimization
BarIterLimit	Integer	Iteration limit of barrier method
MatrixTol	Double	Input matrix coefficient tolerance
FeasTol	Double	The feasibility tolerance
DualTol	Double	The tolerance for dual solutions and reduced cost
IntTol	Double	The integrality tolerance for variables
RelGap	Double	The relative gap of optimization
AbsGap	Double	The absolute gap of optimization

- `TimeLimit`

  Double parameter.

  Time limit of the optimization.

  Default: 1e20

  Minimal: 0

  Maximal: 1e20

- `NodeLimit`

  Integer parameter.

  Node limit of the optimization.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

Maximal: INT_MAX

- BarIterLimit

  Integer parameter.

  Iteration limit of barrier method.

  Default: 500

  Minimal: 0

  Maximal: INT_MAX

- MatrixTol

  Double parameter.

  Input matrix coefficient tolerance.

  Default: 1e-10

  Minimal: 0

  Maximal: 1e-7

- FeasTol

  Double parameter.

  The feasibility tolerance.

  Default: 1e-6

  Minimal: 1e-9.

  Maximal: 1e-4

- DualTol

  Double parameter.

  The tolerance for dual solutions and reduced cost.

  Default: 1e-6

  Minimal: 1e-9

  Maximal: 1e-4

- IntTol

  Double parameter.

  The integrality tolerance for variables.

  Default: 1e-6

  Minimal: 1e-9

  Maximal: 1e-1

- RelGap

  Double parameter.

  The relative gap of optimization.

  Default: 1e-4

  Minimal: 0

  Maximal: DBL_MAX

- AbsGap

Double parameter.

The absolute gap of optimization.

Default: 1e-6

Minimal: 0

Maximal: `DBL_MAX`

10.2 Presolving and scaling

Table 10.2: Presolving and scaling parameters

Name	Type	Description
Presolve	Integer	Level of presolving before solving a model
Scaling	Integer	Whether to perform scaling before solving a problem
Dualize	Integer	Whether to dualize a problem before solving it

- `Presolve`

  Integer parameter.

  Level of presolving before solving a model.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `Scaling`

  Integer parameter.

  Whether to perform scaling before solving a problem.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: No scaling.

  1: Apply scaling.

- `Dualize`

  Integer parameter.

  Whether to dualize a problem before solving it.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: No dualizing.

  1: Dualizing the problem.

10.3 Linear programming related

Table 10.3: Linear programming related parameters

Name	Type	Description
LpMethod	Integer	Method to solve the LP problem
DualPrice	Integer	Specifies the dual simplex pricing algorithm
DualPerturb	Integer	Whether to allow the objective function perturbation when using the dual simplex method
BarHomogeneous	Integer	Whether to use homogeneous self-dual form in barrier
BarOrder	Integer	Barrier ordering algorithm
Crossover	Integer	Whether to use crossover
ReqFarkasRay	Integer	Advanced topic. Whether to compute the dual Farkas or primal ray when the LP is infeasible or unbounded

- `LpMethod`

  Integer parameter.

  Method to solve the LP problem.

  Default: -1

  Possible values:

  -1: Default (Use dual simplex method).

  1: Dual simplex.

  2: Barrier.

  3: Crossover.

  4: Concurrent (Use simplex and barrier simultaneously).

  5: Choose between simplex and barrier automatically (Based on features such as sparsity and/or coefficients ranges).

- `DualPrice`

  Integer parameter.

  Specifies the dual simplex pricing algorithm.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Using Devex pricing algorithm.

  1: Using dual steepest-edge pricing algorithm.

- `DualPerturb`

  Integer parameter.

  Whether to allow the objective function perturbation when using the dual simplex method.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: No perturbation.

  1: Allow objective function perturbation.

- `BarHomogeneous`

  Integer parameter.

  Whether to use homogeneous self-dual form in barrier.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: No.

  1: Yes.

- `BarOrder`

  Integer parameter.

  Barrier ordering algorithm.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Approximate Minimum Degree (AMD).

  1: Nested Dissection (ND).

- `Crossover`

  Integer parameter.

  Whether to use crossover.

  Default: 1

  Possible values:

  -1: Choose automatically.

  Only run crossover when the LP solution is not primal-dual feasible.

  0: No.

  1: Yes.

- `ReqFarkasRay`

  Integer parameter.

  Advanced topic. Whether to compute the dual Farkas or primal ray when the LP is infeasible or unbounded.

  Default: 0

  Possible values:

  0: No.

  1: Yes.

10.4 Semidefinite programming related

- `SDPMethod`

  Integer parameter.

  Method to solve the SDP problem.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Primal-Dual method.

  1: ADMM method.

  2: Dual method.

10.5 Integer programming related

Table 10.4: Integer programming related parameters

Name	Type	Description
CutLevel	Integer	Level of cutting-planes generation
RootCutLevel	Integer	Level of cutting-planes generation of root node
TreeCutLevel	Integer	Level of cutting-planes generation of search tree
RootCutRounds	Integer	Rounds of cutting-planes generation of root node
NodeCutRounds	Integer	Rounds of cutting-planes generation of search tree node
HeurLevel	Integer	Level of heuristics
RoundingHeurLevel	Integer	Level of rounding heuristics
DivingHeurLevel	Integer	Level of diving heuristics
SubMipHeurLevel	Integer	Level of Sub-MIP heuristics
StrongBranching	Integer	Level of strong branching
ConflictAnalysis	Integer	Whether to perform conflict analysis
MipStartMode	Integer	Mode of MIP starts
MipStartNodeLimit	Integer	Limit of nodes for MIP start sub-MIPs

- `CutLevel`

  Integer parameter.

  Level of cutting-planes generation.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `RootCutLevel`

  Integer parameter.

  Level of cutting-planes generation of root node.

  Default: -1

Possible values:

-1: Choose automatically.

0: Off

1: Fast

2: Normal

3: Aggressive

- `TreeCutLevel`

  Integer parameter.

  Level of cutting-planes generation of search tree.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `RootCutRounds`

  Integer parameter.

  Rounds of cutting-planes generation of root node.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

  Maximal: `INT_MAX`

- `NodeCutRounds`

  Integer parameter.

  Rounds of cutting-planes generation of search tree node.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

  Maximal: `INT_MAX`

- `HeurLevel`

  Integer parameter.

  Level of heuristics.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `RoundingHeurLevel`

Integer parameter.

Level of rounding heuristics.

Default: -1

Possible values:

-1: Choose automatically.

0: Off

1: Fast

2: Normal

3: Aggressive

- `DivingHeurLevel`

  Integer parameter.

  Level of diving heuristics.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `SubMipHeurLevel`

  Integer parameter.

  Level of Sub-MIP heuristics.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `StrongBranching`

  Integer parameter.

  Level of strong branching.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `ConflictAnalysis`

  Integer parameter.

  Whether to perform conflict analysis.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: No

  1: Yes

- `MipStartMode`

  Integer parameter.

  Mode of MIP starts.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Do not use any MIP starts.

  1: Only load full and feasible MIP starts.

  2: Only load feasible ones (complete partial solutions by solving subMIPs).

- `MipStartNodeLimit`

  Integer parameter.

  Limit of nodes for MIP start sub-MIPs.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

  Maximal: `INT_MAX`

10.6 Parallel computing related

Table 10.5: Parallel computing related parameters

Name	Type	Description
Threads	Integer	Number of threads to use
BarThreads	Integer	Number of threads used by barrier. If value is -1, the thread count is determined by parameter `Threads`
SimplexThreads	Integer	Number of threads used by dual simplex. If value is -1, the thread count is determined by parameter `Threads`
CrossoverThreads	Integer	Number of threads used by crossover. If value is -1, the thread count is determined by parameter `Threads`
MipTasks	Integer	Number of MIP tasks in parallel

- `Threads`

  Integer parameter.

  Number of threads to use.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

Maximal: 128

- `BarThreads`

  Integer parameter.

  Number of threads used by barrier. If value is -1, the thread count is determined by parameter `Threads`.

  Default: -1

  Minimal: -1

  Maximal: 128

- `SimplexThreads`

  Integer parameter.

  Number of threads used by dual simplex. If value is -1, the thread count is determined by parameter `Threads`.

  Default: -1

  Minimal: -1

  Maximal: 128

- `CrossoverThreads`

  Integer parameter.

  Number of threads used by crossover. If value is -1, the thread count is determined by parameter `Threads`.

  Default: -1

  Minimal: -1

  Maximal: 128

- `MipTasks`

  Integer parameter.

  Number of MIP tasks in parallel.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

  Maximal: 256

10.7 IIS computation related

- `IISMethod`

  Integer parameter.

  Method for IIS computation.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Find smaller IIS.

  1: Find IIS quickly.

10.8 Feasibility relaxation related

- `FeasRelaxMode`

  Integer parameter.

  Method for feasibility relaxation.

  Default: 0

  Possible values:

  0: Minimize sum of violations.

  1: Optimize original objective function under minimal sum of violations.

  2: Minimize number of violations.

  3: Optimize original objective function under minimal number of violations.

  4: Minimize sum of squared violations.

  5: Optimize original objective function under minimal sum of squared violations.

10.9 Parameter Tuning related

Table 10.6: Tuner related parameters

Name	Type	Description
TuneTimeLimit	Double	Time limit for parameter tuning
TuneTargetTime	Double	Time target for parameter tuning
TuneTargetRelGap	Double	Optimal relative tolerance target for parameter tuning
TuneMethod	Integer	Method for parameter tuning
TuneMode	Integer	Mode for parameter tuning
TuneMeasure	Integer	Parameter tuning result calculation method
TunePermutes	Integer	Permutations for each trial parameter set
TuneOutputLevel	Integer	Parameter tuning log output intensity

- `TuneTimeLimit`

  Double parameter.

  Time limit for parameter tuning. A value of 0 indicates that the solver is set automatically.

  Default: 0

  Minimal: 0

  Maximal: 1e20

- `TuneTargetTime`

  Double parameter.

  Time target for parameter tuning.

  Default: 1e-2

  Minimal: 0

  Maximal: `DBL_MAX`

- `TuneTargetRelGap`

Double parameter.

Optimal relative tolerance target for parameter tuning.

Default: 1e-4

Minimal: 0

Maximal: `DBL_MAX`

- `TuneMethod`

  Integer parameter.

  Method for parameter tuning.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Greedy search strategy.

  1: Broader search strategy.

- `TuneMode`

  Integer parameter.

  Mode for parameter tuning.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Solving time.

  1: Optimal relative tolerance.

  2: Objective function value.

  3: The lower bound of the objective function value.

- `TuneMeasure`

  Integer parameter.

  Parameter tuning result calculation method.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Calculate the average value.

  1: Calculate the maximum value.

- `TunePermutes`

  Integer parameter.

  Permutations for each trial parameter set. A value of 0 indicates that the solver is automatically set.

  Default: 0

  Minimal: 0

  Maximal: `INT_MAX`

- `TuneOutputLevel`

Integer parameter.

Parameter tuning log output intensity.

Default: 2

Possible values:

0: No output of tuning log.

1: Output only a summary of the improved parameters.

2: Output a summary of each tuning attempt.

3: Output a detailed log of each tuning attempt.

10.10 Callback related

- `LazyConstraints`

  Integer parameter.

  Whether lazy constraints are part of the model.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: No.

  1: Yes.

Notes

- This parameter only affects MIP.

10.11 Other parameters

- `Logging`

  Integer parameter.

  Whether to print optimization logs.

  Default: 1

  Possible values:

  0: No optimization logs.

  1: Print optimization logs.

- `LogToConsole`

  Integer parameter.

  Whether to print optimization logs to console.

  Default: 1

  Possible values:

  0: No optimization logs to console.

  1: Print optimization logs to console.

10.12 Methods for accessing and setting parameters

In different programming interfaces, the ways to access and set information are slightly different. For details, please refer to the corresponding chapters of each programming language API reference manual:

- C API: *C API Functions: Accessing and setting parameters*
- C++ API: *C++ API Reference: Parameters*
- C# API: *C# API Reference: Parameters*
- Java API: *Java API Reference: Parameters*
- Python API: *Python API Reference: Parameters*

Chapter 11

MIP Starts

11.1 Utilities of MIP Starts

11.1.1 Set and Load MIP Starts

For MIP problems, COPT provides methods to specify initial solution value(s) for a single variable or set of variables and load it/them into model. The parameters that can be specified are:

- `vars` :variables
- `startvals` :variables' solution values

The functions in different APIs are shown in Table 11.1:

Table 11.1: Functions for setting MIP starts

API	Function
C	`COPT_AddMipStart`
C++	`Model::SetMipStart()`
C#	`Model.SetMipStart()`
Java	`Model.setMipStart()`
Python	`Model.setMipStart(vars, startvals)`

Note

- Regarding the operations of MIP starts, their **function names**, **calling methods**, and **parameter names** are slightly different in different programming interfaces, but the implementation of functions and meanings of parameter are the same.
- For more details on setting initial solutions in C, please refer to funtion `COPT_AddMipStart` in chapter *C API Function: MIP start utilities*.
- You may want to call this method several times to input the MIP start. Please call `loadMipStart()` once when the input is done.

11.1.2 Read and Write MIP Starts

COPT provides functions for file read/write. It can read variable values from a MIP start file (`".mst"`) as the initial solution values of variables, and write the solving results or existing initial solution to a MIP start file (`".mst"`). The functions for reading and writing MIP start file in different APIs are shown in Table 11.2:

Table 11.2: Functions for reading and writing MIP starts

API	read MIP starts	write MIP starts
C	`COPT_ReadMst`	`COPT_WriteMst`
C++	`Model::ReadMst`	`Model::WriteMst`
C#	`Model.ReadMst()`	`Model.WriteMst()`
Java	`Model.readMst()`	`Model.writeMst()`
Python	`Model.readMst()`	`Model.writeMst()`

11.2 Related Parameters

COPT provides the following parameters that control how MIP starts (initial solutions) are handled inside.

- `MipStartMode`

  Integer parameter.

  Mode of MIP starts, i.e. how MIP starts are handled.

  Default -1

  Possible values:

  -1: Automatic.

  0: Do not use any MIP starts.

  1: Only load full and feasible MIP starts.

  2: Only load feasible ones (complete partial solutions by solving subMIPs).

Note: If the provided initial solution is incomplete (partial), `MipStartMode=2` needs to be set, otherwise the initial solution will be rejected.

- `MipStartNodeLimit`

  Integer parameter.

  Limit of nodes for MIP start sub-MIPs.

  Default: -1

  Minimal: -1

  Maximal: `INT_MAX`

11.3 Log of MIP starts

11.3.1 MIP starts are accepted

1.A (better) initial solution was provided

```
Initial MIP solution # 1 with objective value 9.73987 was accepted
```

2.A partial initial solution was provided, and set `MipStartMode=2`. (complete it by solving a subMIP)

```
Loading 1 initial MIP solution
Extending partial MIP solution # 1
Extending partial MIP solution # 1 succeed (0.2s)
Initial MIP solution # 1 with objective value 9.66566 was accepted
```

11.3.2 MIP starts are rejected

1.The provided initial solution was infeasible

```
Initial MIP solution # 1 was rejected: Primal Inf 1.00e+00  Int Inf 1.78e-15
```

2.The provided initial solution was not better than the current best one

```
Initial MIP solution # 2 with objective value 10.3312 was rejected (not better)
```

3.The provided initial solution was incomplete (partial), and not set MipStartMode=2

```
Loading 1 initial MIP solution
Initial MIP solution # 1 was rejected: partial
```

4.The provided initial solution was incomplete (partial), and COPT failed to find a feasible solution by solving subMIP

```
Loading 1 initial MIP solution
Extending partial MIP solution # 1
Extending partial MIP solution # 1 failed (infeasible)
Initial MIP solution # 1 was rejected: partial
```

Chapter 12

MIP Solution Pool

In general, the solver finds multiple feasible solutions in the process of solving a MIP problem with Branch-and-cut method. COPT provides a solution pool for MIP problem, from which users can obtain solutions and the corresponding objective function values.

COPT provides functions that users can get the `iSol` th solution's objective function value and solution values (of specified variables) by specifying the following parameters.

- `iSol`: Index of the solution to obtain. (0-based)
- `vars`: Variables

The functions in different APIs are shown in Table 12.1:

Table 12.1: Get solutions and objective values from solution pool

API	Get solution	Get objective value
C	`COPT_GetSolution`	`COPT_GetPoolObjVal`
C++	`Model::GetPoolSolution()`	`Model.GetPoolSolution()`
C#	`Model.GetPoolSolution()`	`Model.GetPoolSolution()`
Java	`Model.getPoolSolution()`	`Model.getPoolObjVal()`
Python	`Model.getPoolSolution()`	`Model.getPoolObjVal()`

Note: Regarding the operations of solution pool, their **function names**, **calling methods**, and **parameter names** are slightly different in different programming interfaces, but the implementation of functions and meanings of parameter are the same.

Attributes of Solution Pool

- `PoolSols`

  Integer attribute

  Number of solutions in the solution pool.

Chapter 13

COPT Tuner

13.1 Introduction

The COPT Tuner is a tool designed for tuning performance automatically for all supported problem types.

- For MIP problems, it supports tuning for solving time, relative gap, best objective value and objective bound;
- For non-MIP problems, only solving time supported.

The workflow of the COPT tuning tool is as follows:

1. First, perform benchmark calculation and allow users to customize benchmark calculation parameters;
2. Next, generate tuning parameters one by one, and find parameter combinations that improve the solution performance through parameter tuning calculations.

13.2 Related parameters

Firstly, the tuner will do a baseline run, possibly with fixed parameters from users, then move to the improvement run, where the tuner will generate trial parameter sets and search for parameters that can improve the performance. To summarize, the COPT Tuner provides the following capabilities:

Table 13.1: Tuner related parameters

Name	Type	Description
TuneTimeLimit	Double	Time limit for parameter tuning
TuneTargetTime	Double	Time target for parameter tuning
TuneTargetRelGap	Double	Optimal relative tolerance target for parameter tuning
TuneMethod	Integer	Method for parameter tuning
TuneMode	Integer	Mode for parameter tuning
TuneMeasure	Integer	Parameter tuning result calculation method
TunePermutes	Integer	Permutations for each trial parameter set
TuneOutputLevel	Integer	Parameter tuning log output intensity

13.3 Provided capabilities

The COPT tuning tool provides the following capabilities:

13.3.1 Tuning method

Controlled by the parameter `TuneMethod`, options are: greedy search and aggressive search. The greedy method tries to find better parameter settings within limited number of trials, while the aggressive method search for more combinations and has much larger search space than the greedy one, and can potentially find even better parameter settings at the expense of more elapsed tuning time. Default setting is to choose automatically.

- Greedy search method: It is expected to optimize the calculation with fewer parameters and find better parameter settings;
- Extensive search method: try more parameter combinations, have a larger search space, and are more likely to find better parameter settings, but also consume more tuning time.

The possible values and corresponding meanings of the parameter `TuneMethod` are as follows. By setting it to a different value, the search method can be selected. The default setting is automatic selection.

- -1: automatic selection
- 0: Greedy search strategy
- 1: Broader search strategy

13.3.2 Tuning mode

Controlled by the parameter `TuneMode`, options are: solving time, relative gap, objective value and objective bound. For MIP problem, by default, if the baseline run is not solved to optimality within specified time limit, tuner will change tuning mode to relative gap automatically. Default setting is to choose automatically.

- 0: Solving time
- 1: Optimal relative tolerance
- 2: Objective function value
- 3: Lower bound of objective function value

Note: For integer programming problems, by default, if the benchmark calculation does not optimize the model within the given time limit, the tuning tool will automatically switch the tuning mode to the optimal relative tolerance .

13.3.3 Tuning permutations

Controlled by the parameter `TunePermutes`. Tuner allow users to run more permutations for each trial parameter set to evaluate performance variability. Default setting is to choose automatically.

13.3.4 Tuning measure

Controlled by the parameter `TuneMeasure`, options are: by average or maximum. When users run more permutations for each trial, tuner will compute the aggregated tuning value by this measure. Default setting is to choose automatically.

- 0: Calculate the average
- 1: Calculate the maximum value

13.3.5 Tuning targets

Controlled by the parameter `TuneTargetTime` and `TuneTargetRelGap`. Tuner enables users to specify target solving time or relative gap for tuning, when tuner finds out parameters that satisfy the specified target, it will stop tuning. For solving time, default value is 0.01 seconds, while for relative gap, default value is 1e-4.

13.3.6 Tuning output

Controlled by the parameter `TuneOutputLevel`, options are: no output, show summary for improved trials, show summary for each trial and show detailed log for each trial. Default setting is to show summary for each trial.

- 0: Do not output tuning log
- 1: Output only a summary of the improved parameters
- 2: Output a summary of each tuning attempt
- 3: Output a detailed log of each tuning attempt

13.3.7 TuneTimeLimit

Controlled by the parameter `TuneTimeLimit`. This parameter is used to control the overall time limit for the improvement run of tuning. Default setting is to choose automatically.

13.3.8 User defined parts

- User defined parameters

  The tool enables users to set parameters for the baseline run, which will also be used as fixed parameters for each trial run. Tuner will not tune parameters in the fixed parameters.

- User defined MIP start

  The COPT tuner enables users to set MIP start for the baseline run, which will also be used for each trial run also.

- User defined tuning file

  The COPT tuner enables users to read tuning parameter sets from tuning file, if so, the tuner will try to tune from the given parameter sets, otherwise, tuner will generate tuning parameter sets automatically. The tuning file is similar to parameter file, with the difference that it allow multiple values for each parameter name.

  Note: The COPT tuning file has a similar format to the COPT parameter file, except that the tuning file allows multiple values to be specified for a single parameter.

13.3.9 Load or writing tuning parameter

After the parameter tuning is completed, the number of parameter tuning results can be obtained through the attribute `TuneResults`, and the tuning results of the specified number can also be loaded into the model or written into the parameter file.

COPT can output the parameter tuning results of the specified number to the parameter file (`".par"`). The parameters that need to be specified are:

- `idx`: parameter tuning result number
- `filename`: file name

The corresponding functions in different programming interfaces are as follows:

Table 13.2: functions for writing parameter tuning results in different interfaces

API	function
C	`COPT_WriteTuneParam`
C++	`Model::WriteTuneParam()`
C#	`Model.WriteTuneParam()`
Java	`Model.writeTuneParam()`
Python	`Model.writeTuneParam()`

13.4 Example

For example, to tune model `"foo.mps"` from command line for solving time with COPT command line tool, the commands are:

```
copt_cmd -c "read foo.mps; tune; exit"
```

To use the tuner in API such as Python, the codes are:

```
env = Envr()
m = env.createModel()
m.read("foo.mps")
m.tune()
```

Chapter 14

Callback

COPT provides the Callback utility, which supports users to obtain intermediate information (such as the current best bound, the current optimal objective value, etc.) during the MIP solving process; or control the solving process: such as modifying the model (such as adding lazy constraints, adding cutting planes), terminating the solving procress and so on.

The content of this chapter is organized as follows:

- *Get the intermediate information of the MIP*
- *Control the MIP solving process*
- *Callback function used in different APIs*

14.1 Obtain the intermediate information of the MIP

The intermediate information that can be obtained during the MIP solving process depends on the Callback Context. The supported Contexts are as follows:

- `CBCONTEXT_MIPSOL`: Invokes the callback when a new MIP candidate solution is found.
- `CBCONTEXT_MIPRELAX` : Invokes the callback when a new LP-relaxation solution is found.

Regarding the Callback information supported by COPT, please refer to the *Callback related information* section for details.

The corresponding relationships are listed in the following table:

Context	Callback Information
`CBCONTEXT_MIPSOL`	`MipCandObj`, `MipCandidate`
`CBCONTEXT_MIPRELAX`	`RelaxSolution`, `RelaxSolObj`

Besides, `BestObj`, `BestBnd`, `HasIncumbent`, `Incumbent` can be obtained at any context (that is, both `CBCONTEXT_MIPSOL` and `CBCONTEXT_MIPRELAX`).

Notes

1. If `HasIncumbent == False`, then `Incumbent` cannot be obtained.
2. For **Incumbent** , **LP-relaxation Solution** , **Incumbent**, these three items are obtained through different methods in different interfaces:
    - C language: through the function `COPT_GetCallbackInfo`, the name of the intermediate information to be obtained is provided as a parameter of the function;

- In object-oriented programming languages (C++/C#/Java/Python), the `CallbackBase` class provides specialized functions to obtain corresponding intermediate information.

3. In object-oriented programming languages (C++/C#/Java/Python), `CallbackBase.getInfo(infoname)` function, only supports obtaining information constants of scalar type, namely: `MipCandObj`, `RelaxSolObj`, `BestObj`, `BestBnd`, `HasIncumbent`. As for other non-scalar information, they provide specialized function access.

   Taking the Python interface as an example, the corresponding function names are listed below. Other programming language interfaces are similar, you can refer to the `CallbackBase` class of each API.

   - Get the current feasible solution: `CallbackBase.getSolution()`
   - Get the current LP relaxation solution: `CallbackBase.getRelaxSol()`
   - Get the current optimal feasible solution: `CallbackBase.getIncumbent()`

14.2 Controlling the MIP solving process

COPT provides functions to allow users to interactively add lazy constraints or cutting planes during the solving process of the MIP branch-and-cut to control the MIP solving process. There are three main types of operations:

1. Add lazy constraints
2. Add cutting planes
3. Set up heuristic solutions

14.2.1 Add lazy constraints

COPT supports users to add lazy constraints in two ways, one is to add directly to the model, and the other is to add in specified context during the solving process through Callback. In the C API, it is distinguished according to whether the function name contains `"Callback"`; In the object-oriented API, the functions corresponding to `Model` class and `CallbackBase` respectively:

1. Before solving, user can directly add to the model, and COPT supports adding unilateral and bilateral lazy constraints. Take Python as an example:

- Add a single lazy contraint: Model.addLazyConstr()
- Add a set of lazy contraints: Model.addLazyConstrs()

2. During the solving process, user can dynamically add according to the specified context through callback, and COPT only supports the addition of single-sided lazy constraints. Take Python as an example:

- Add a single lazy contraint: CallbackBase.addLazyConstr()
- Add a set of lazy contraints: CallbackBase.addLazyConstrs()

14.2.2 Add cutting planes

Similarly, for cutting planes, COPT also supports the above two adding methods.

1. Before solving, user can directly add to the model, and COPT supports adding unilateral and bilateral cutting-planes. Take Python as an example:

- Add a single cutting-plane: Model.addUserCut()
- Add a set of cutting-planes: Model.addUserCuts()

2. During the solving process, user can dynamically add according to the specified context through callback, and COPT only supports the addition of single-sided cutting-planes. Take Python as an example:

- Add a single cutting-plane: CallbackBase.addUserCut()
- Add a set of cutting-planes: CallbackBase.addUserCuts()

---

note

- It is invalid to call the functions of `Model` class to add lazy constraints or user cutting-planes in Callback.

14.2.3 Set heuristic solutions

COPT supports adding heuristic solutions during the MIP solving process.

- Set the heuristic solution: `CallbackBase.setSolution(vars, val)`
- Load a custom solution into the model: `CallbackBase.loadSolution()`

---

note

Currently supports setting complete feasible solutions.

Similarly, certain callback functions that operate on models can only be called at specified context.

Taking python as an example, the functions in `CallbackBase` class and the corresponding callback context are listed as follows:

Context	Function

CBCONTEXT_MIPSOL	`CallbackBase.addLazyConstr`
	`CallbackBase.addLazyConstrs`
	`CallbackBase.getSolution`
	`CallbackBase.setSolution`
	`CallbackBase.getIncumbent`
	`CallbackBase.getInfo`

CBCONTEXT_MIPRELAX	`CallbackBase.addUserCut`
	`CallbackBase.addUserCuts`
	`CallbackBase.addLazyConstr`
	`CallbackBase.addLazyConstrs`
	`CallbackBase.getRelaxSol`
	`CallbackBase.getIncumbent`
	`CallbackBase.setSolution`
	`CallbackBase.getInfo`

---

note

The function forms may slightly look different in different APIs, but the corresponding relationship is the same. For the C language chapter, please refer to *C API: callback function* for details.

14.3 Using the callback utilities in different APIs

Take the object-oriented programming interface as an example to show the basic steps of calling the Callback function in different APIs:

1. Build a custom Callback class and inherit the `CallbackBase` class;
2. Implement the `CallbackBase.callback()` function, in which you can customize the operations you need to perform (such as obtaining intermediate information or adding lazy constraints/cutting-planes);
3. Create a new custom Callback instance and pass in the arguments required;
4. Add a Callback instance through `Model.setCallback()` of the Model class, and input the Callback Context as a parameter.

In the subsequent solving process, in the specified Callback Context, the user-defined callback function in the Callback instance will be called.

The specific implementation in each programming language API is also similar, you can refer to the sample code.Taking Python as an example, `"cb_ex1.py"` is in the examples directory in the installation package.

The calling method and function name of the callback function in different programming interfaces are **slightly different**, but the supported functions and function meanings are the same. Please refer to the corresponding chapters of different programming interface API reference manuals for specific introductions:

- C API: *callback function*
- C++ API: *CallbackBase class*
- C# API: *CallbackBase class*
- Java API: *CallbackBase class*
- Python API: *CallbackBase class*

Chapter 15

Matrix Modeling Method

The COPT Python API provides matrix modeling, supports `NumPy` multi-dimensional array, a two-dimensional NumPy matrix, SciPy compressed sparse column matrix (`csc_matrix`) and compressed sparse row matrix (`csr_matrix`) operations (`NumPy` minimum version requirement is 1.23, `Python` minimum version requirement is 3.8) and can be combined with ordinary (scalar) variables and constraints. COPT mainly provides the following utilities:

1. Add multi-dimensional variables (`MVar`) and other related operations;
2. Construct multi-dimensional linear expressions (`MLinExpr`), add multi-dimensional linear constraints (`MConstr`) and other related operations;
3. Construct multi-dimensional quadratic expression (`MQuadExpr`), add multi-dimensional convex quadratic constraint (`QConstraint`) and other related operations.

15.1 Multi-dimensional Variables

1. Add multi-dimensional variable `MVar`

   `MVar` constains operations related to multi-dimensional variables. Users can use `Model.addMVar()` to add a matmulti-dimensionalrix variable `MVar` of any dimension and shape to the model. In addition to the need of specifying the argument `shape` (matrix shape), the rest of the arguments are consistent with ordinary variables, including: `lb` , `ub` , `vtype` , `nameprefix` .

   - Add one-dimensional continuous multi-dimensional variables: `x = Model.addMVar(3)`
   - Add two-dimensional 3x3 binary multi-dimensional variables: `y = Model.addMVar(shape(3,3), vtype=COPT.BINARY)`

   In addition, the `MVar` multi-dimensional variables can also be sliced, such as: `y1 = y[:,0:2]`
2. Get multi-dimensional variable related attributes:
   - Number of dimensions: `MVar.ndim`
   - The shape of the multi-dimensional variable: `MVar.shape`
   - Number of elements in the multi-dimensional variable: `MVar.size`

15.2 Multi-dimensional array operations and expressions

15.2.1 Multi-dimensional Linear Expressions

Multi-dimensional variables and their coefficients (can be `ndarray`) form a Multi-dimensional linear expression (`MLinExpr`), and the supported operations mainly include:

1. Matrix multiplication: A @ x

```
x = model.addMVar(3)
A = np.array([[1, 0, 1],[0, 0, 1]])
expr1 = A @ x
```

2. Vector inner product

```
x = model.addMVar(3)
c = np.array([1, 2, 3])
expr2 = c @ x
```

15.2.2 Multi-dimensional Quadratic Expression

Common multi-dimensional quadratic expressions and their corresponding mathematical forms are as follows:

- x @ Q @ x: $x^T Q x$
- x @ x: $x^T x$
- x @ Q @ x + c @ x + b: $x^T Q x + c^T x + b$

15.2.3 Other multi-dimensional array operations

1. Combine with regular linear variables, regular linear expressions, and constants:

```
x = model.addMVar(3)
y = model.addVar()
c = np.array([1, 2, 3])
Q = np.full((3, 3), 1)
expr3 = 2 * x @ Q @ x + c @ x + 2 * y + 1
```

2. Self-increment/self-subtraction/self-multiplication operations:

```
mx = m.addMVar((3, 3))
B = np.array([[1, 0, 1], [0, 1, 1]])
expr_add = B @ mx
expr_add += 1
expr_add *= 2
```

Notes

- When we directly print the multi-dimensional expression with `print(MLinExpr)`/ `print(MQuadExpr)` , the `shape` of the expression will be output at the same time. When `shape=()` , it means that the expression is a scalar (single linear/quadratic expression), corresponding to `ndim=0` , `size=1` . The same is true for multi-dimensional variables `MVar` ;
- When performing matrix multiplication (A@x), the matrix multiplication algorithm needs to be satisfied, and the number of columns of A and the number of rows of X need to be the same;

- COPT supports the combination of `MLinExpr` and `LinExpr` , but it should be noted that the `MLinExpr` needs `shape=()` at this time, and the final returned expression is `MLinExpr` with `shape=()` .

15.3 Matrix Constraints

15.3.1 Matrix linear Constraints

COPT supports two ways of adding multi-dimensional linear constraints, and the format provided by the function arguments is different:

1. `Model.addMConstr()` that specifically adds multi-dimensional linear constraints, the arguments that can be specified are:
   - `A` : coefficient matrix for linear constraints
   - `x` : decision variables (`MVar`)
   - `sense` : type of linear constraint, the possible values are: `'L'` (<=), `'G'` (>=), `'E'` (=)
   - `b` : right-hand-side of linear constraints (vector with dimensions equal to the number of rows of matrix `A`)
   - `name` : name prefix for linear constraints

```
x = model.addMVar(shape=3, vtype=COPT.BINARY, nameprefix='x')
A = np.array([[1, 2, 3], [3, 2, 1]])
b = np.array([2, 5])
mconstrs = model.addMConstr(A, x, 'L', b, nameprefix='c')
obj = np.array([1, 2, 1])
model.setObjective(obj @ x, COPT.MINIMIZE)
```

2. Matrix linear constraints can be regarded as a set of linear constraints, so `Model.addConstrs()` can also add multi-dimensional linear constraints:

```
x = model.addMVar(shape=3, vtype=COPT.BINARY, nameprefix='x')
A = np.array([[1, 2, 3], [3, 2, 1]])
b = np.array([2, 5])
mconstrs = model.addConstrs(A @ x <= b, nameprefix='c')
obj = np.array([1, 2, 1])
model.setObjective(obj @ x, COPT.MINIMIZE)
```

15.3.2 Quadratic Constraints

COPT supports two ways of constructing multi-dimensional quadratic constraints, and the format provided by the function arguments is different:

1. `Model.addMQConstr()` that specifically adds multi-dimensional quadratic constraints, the arguments that can be specified are:
   - `Q` : quadratic coefficient matrix
   - `c` : vector of linear term coefficients, or `None` if there is no linear term
   - `sense` : type of quadratic constraint, the possible values are: `'L'` (<=), `'G'` (>=), `'E'` (=)
   - `rhs` : right-hand-side of quadratic constraints
   - `xQ_L` : the left-hand variable of the quadratic coefficient matrix Q (vector whose length is consistent with the number of rows of the matrix `Q`)

- `xQ_R` : right-hand variable of the quadratic coefficient matrix Q (vector whose length is consistent with the number of columns of the matrix `Q`)
- `xc` : the variables for the linear term, or `None` if there is no linear term
- `name` : name prefix for quadratic constraints

```
Q = np.diag([3, 2, 1])
x = model.addMVar(3)
y = model.addMVar(3)
c1 = model.addMQConstr(Q, None, 'E', 1.0, x, y)
```

2. `Model.addQConstr()`, directly gives the multi-dimensional quadratic expression
   - `lhs` : multi-dimensional quadratic expression
   - `sense` : constraint type
   - `rhs` : right-hand-side of quadratic constraints

```
Q = np.diag([3, 2, 1])
x = model.addMVar(3)
y = model.addMVar(3)
c2 = model.addQConstr(lhs=x@Q@y, sense=COPT.EQUAL, rhs=1.0)
```

15.4 Objective function composed of multi-dimensional variables

COPT supports setting linear and quadratic objective functions, and provides two ways to set objective functions. The format of function arguments is different:

1. `Model.setMObjective()` that specifically sets the objective function composed of multi-dimensional variables, the arguments that can be specified are:
   - `Q` : quadratic coefficient matrix, or `None` if the objective function is linear
   - `c` : vector of linear term coefficients, or `None` if there is no linear term
   - `constant` : the constant term of the objective function
   - `xQ_L`: the left-hand variable of the quadratic term coefficient matrix Q (vector whose length is consistent with the number of rows of the matrix `Q`), or `None` if the objective function is linear
   - `xQ_R`: the right-hand variable of the quadratic coefficient matrix Q (vector, whose length is consistent with the number of columns in the matrix `Q`), or `None` if the objective function is linear
   - `xc`: the variable for the linear term, or `None` if there is no linear term
   - `sense`: direction of optimization, possible values are: `COPT.MINIMIZE` or `COPT.MAXIMIZE`
2. `Model.setObjective()` that directly gives the expression of the objective function
   - `expr`: Objective function expression, which can be linear or quadratic
   - `sense`: optimization direction, possible values are: `COPT.MINIMIZE` or `COPT.MINIMIZE`

```
x = model.addMVar(shape=3, vtype=COPT.BINARY, nameprefix="x")
obj = np. array([1, 2, 1])
model.setObjective(obj @ x, COPT.MINIMIZE)
```

Regarding the matrix modeling method, the COPT Python interface provides multi-dimensional variables, (linear and convex quadratic) expressions, and matrix constraint classes respectively, and contains related operations. For the methods and specific introductions included, please refer to the corresponding part of the Python API :

- Multi-dimensional variable: *MVar*
- Multi-dimensional linear expression: *MLinExpr*
- Multi-dimensional quadratic expressions: *MQuadExpr*
- Matrix constraints: *MConstr*

Chapter 16

FAQs

16.1 Installation and Licensing Configuration Related

- **Q:** What is the reason for the error `invalid username` when configuring the license?

  A: This error indicates that the username was incorrectly filled in when applying, and you can re-fill the form with the correct username to apply. For information on how to obtain username under different operating systems, please refer to the Solver COPT application page, please remark in the application reason with **"The username is incorrectly filled in, reapply"** . we will issue new license for the correct username.

- **Q:** After downloading COPT, an antivirus software installed on the computer reports a virus and automatically isolates it.

  A: The COPT software downloaded from the COPT official download link is the official version, which has not been developed with any suspicious virus behavior. It can be determined that the anti-virus software is falsely reporting. Please temporarily close the anti-virus software before downloading it.

- **Q:** After validating the license (executing `copt_licgen -v`), it reports an error: `Missing Files` or `Invalid Signature` .

  A: This type of error indicates that the license file configuration fails. Please refer to *Installation Instructions: Configuring License File* to check whether the steps for configuring the license file are correctly followed. Common reasons are as follows:

  1. The license file in the current working directory is not compatible with the version of COPT (for example: the license is version 4.0, while the COPT is version 5.0), please check the `VERSION` in `"license.dat"` to confirm whether the major version is the same, if not, please go to the Solver COPT application page to re-apply, and we will issue you the latest license.
  2. In Windows system, if the COPT software is installed in the system disk (usually `C`) in a non-user directory (eg: the default installation path `"C:\Program Files\copt65"`) , you need to **open the command line window with administrator privileges** and execute the license acquisition command `copt_licgen` again.

- **Q:** I have already installed an old version of the COPT Python interface (`coptpy`), how do I upgrade to the new version?

  A: Please enter: `pip install --upgrade coptpy` in the terminal.

- **Q:** In the Python IDE, when debugging, a variable `not defined` warning is reported, or a wavy line appears below the variable in the code, what is the reason?

  A: Currently `coptpy` supports type hints, please enter `pip install coptpy-stubs` in the terminal to download and install `coptpy-stubs` to solve this problem. After successfully installed, in the

IDE when writing code in, you will be prompted for variable name completion and possible values for function parameters.

16.1.1 MacOS System

- **Q:** When calling `coptpy` on MacOS, an error is reported: `ImportError: from .coptpywrap import * symbol not found in flat namespace`.

  A: This error is due to the mismatch between `coptpy` and Anaconda's architecture on MacOS. Such as `coptpy` is the M1 version, and Anaconda is the x86 version, you can install the M1 version of Anaconda to solve this problem.

- **Q:** When the license is configured in the MacOS system, the `copt_licgen` command is executed in the terminal, and an error is reported: `command not found: copt_licgen` .

  A: This error is because the relevant environment variables of COPT are not configured. In the MacOS system, you need to configure the environment variables after installing COPT. Please refer to Installation Instructions: MacOS System chapter to obtain detailed installation instructions.

- **Q:** When manually configuring environment variables, I copy content directly from the document to `.zshrc` file or `.bash_profile` file, causing the configuration to fail.

  A: Due to the document encoding problem, the above environment variables cannot be directly copied to the corresponding file, and the contents of the environment variables need to be manually entered.

16.1.2 Windows System

- **Q:** In Windows system, when executing `copt_licgen` to generate the license file, an error is reported that the license file cannot be written to the hard disk, and the error message is: `error opening file` .

  A: If the COPT software is installed in the system disk (usually `C`) in a non-user directory (eg: the default installation path `"C:\Program Files\copt65"`), y you need to **open the command line window with administrator privileges** and execute the license acquisition command `copt_licgen`, in order to normally write the license file to the `C` drive. Administrator privileges are not required to execute permission acquisition commands under user directories such as `"C:\Users\shanshu"`.

- **Q:** In Windows system, when installing COPT Python interface via command `pip install coptpy`, an error is displayed: `could not find a version, no matching distribution` , what is the reason?

  A: Please do not use Python installed through Microsoft Store, it is recommended to download from Anaconda distribution or Python official distribution Download Python.

- **Q:** In Windows system, when installing COPT Python interface through COPT installation package (`python setup.py install`), an error `could not create build: access denied` is reported.

  A: If COPT is installed in the system disk (usually `C`) in a non-user directory (eg: the default installation path `"C:\Program Files\copt65"`), you need to first **Open the command line window with administrator privileges** and execute the command `python setup.py install`.

16.2 Modeling and Solving Functions Related

- **Q:** How to deal with the situation where the model is infeasible?

  A: COPT provides functions to calculate IIS and feasible relaxation to analyze the reasons for model infeasibility: Computing IIS will obtain the minimum set of infeasible constraints and variables, and feasibility relaxation attempts to make the model feasible with minimal changes. Please see the Infeasible Model Handling chapter for details.

- **Q:** What is the reason for the error `ValueError: cannot create object arrays from iterator.` when adding matrix variables using the matrix modeling method of COPT?

  A: The matrix modeling function supported by COPT Python has a minimum version requirement, the minimum version requirement for `NumPy` is 1.23, and the minimum version requirement for `Python` is 3.8 . `NumPy` can be upgraded to the latest version by `pip install --upgrade numpy`.

Chapter 17

C API Reference

The **Cardinal Optimizer** provides a C API library for advanced usage. This section documents all the COPT constants, API functions, parameters and attributes listed in `copt.h`.

17.1 Constants

There are three types of constants.

1. Constructing models, such as optimization directions, constraint senses or variable types.
2. Accessing solution results, such as API return code, basis status and LP status.
3. Monitoring optimization progress, such as callback context.

17.1.1 Optimization directions

For different optimization scenarios, it may be required to either maximize or minimize the objective function. There are two optimization directions:

- `COPT_MINIMIZE`

  For minimizing the objective function.

- `COPT_MAXIMIZE`

  For maximizing the objective function.

The optimization direction is automatically set when reading in a problem from file. It can also be set explicitly using `COPT_SetObjSense`.

17.1.2 Infinity

In COPT, the infinite bound is represented by a large value, which can be set using the double parameter `COPT_DBLPARAM_INFBOUND`, whose default value is also available as a constant:

- `COPT_INFINITY`

  The default value (`1e30`) of the infinite bound.

17.1.3 Undefined Value

In COPT, the undefined value is represented by another large value. For example, the default solution value of MIP start is set to a constant:

- `COPT_UNDEFINED`

  Undefined value(`1e40`).

17.1.4 Constraint senses

NOTE: Using constraint senses is supported by COPT but not recommended. We recommend defining constraints using explicit lower and upper bounds.

Traditionally, for optimization models, constraints are defined using **senses**. The most common constraint senses are:

- `COPT_LESS_EQUAL`

  For constraint in the form of $g(x) \leq b$

- `COPT_GREATER_EQUAL`

  For constraint in the form of $g(x) \geq b$

- `COPT_EQUAL`

  For constraint in the form of $g(x) = b$

In additional, there are two less used constraint senses:

- `COPT_FREE`

  For unconstrained expression

- `COPT_RANGE`

  For constraints with both lower and upper bounds in the form of $l \leq g(x) \leq u$.

  Please refer to documentation of `COPT_LoadProb` regarding how to use `COPT_RANGE` to define a constraints with both lower and upper bounds.

17.1.5 Variable types

Variable types are used for defining whether a variable is continuous or integral.

- `COPT_CONTINUOUS`

  Non-integer continuous variables

- `COPT_BINARY`

  Binary variables

- `COPT_INTEGER`

  Integer variables

17.1.6 SOS-constraint types

SOS constraint (Special Ordered Set) is a kind of special constraint that places restrictions on the values that a set of variables can take.

COPT currently support two types of SOS constraints, one is SOS1 constraint, where at most one variable in the constraint is allowed to take a non-zero value, the other is SOS2 constraint, where at most two variables in the constraint are allowed to take non-zero value, and those non-zero variables must be contiguous. Variables in SOS constraints are allowed to be continuous, binary and integer.

- `COPT_SOS_TYPE1`

  SOS1 constraint

- `COPT_SOS_TYPE2`

  SOS2 constraint

17.1.7 Indicator constraint

Indicator constraint is a kind of logical constraints, is uses a binary variable y as the indicator variable, and implies whether the linear constraint $a^T x \leq b$ is valid based on value of variable y. The canonical form of an indicator constraint is:

$$y = f \rightarrow a^T x \leq b \tag{17.1}$$

Where $f \in \{0, 1\}$. If $y = f$, the linear constraint is valid. Otherwise if $y \neq f$, the linear constraint is invalid (may be violated). The sense of the linear constraint can be $\leq$, $\geq$ and $=$.

17.1.8 Second-Order-Cone constraint

The Second-Order-Cone (SOC) constraint is a special type of quadratic constraint, currently, COPT support two types of SOC constraint, which includes:

- `COPT_CONE_QUAD` : Regular cone

$$Q^n = \left\{ x \in \mathbb{R}^n \;\middle|\; x_0 \geq \sqrt{\sum_{i=1}^{n-1} x_i^2}, x_0 \geq 0 \right\} \tag{17.2}$$

- `COPT_CONE_RQUAD` : Rotated cone

$$Q_r^n = \left\{ x \in \mathbb{R}^n \;\middle|\; 2x_0 x_1 \geq \sum_{i=2}^{n-1} x_i^2, x_0 \geq 0, x_1 \geq 0 \right\} \tag{17.3}$$

17.1.9 Quadratic objective function

Besides linear objective function, COPT also supports general convex quadratic objective function.

The mathematical form is:

$$x^T Q x + c^T x \tag{17.4}$$

Where, x is an array of variables, Q is the quadratic part of the quadratic objective funtion and c is the linear part.

17.1.10 Quadratic constraint

Besides the special type of quadratic constraint, Second-Order-Cone (SOC) constraint, COPT also supports general convex quadratic constraint.

The mathematical form is:

$$x^T Q x + q^T x \leq b \tag{17.5}$$

Where, x is an array of variables, Q is the quadratic part of the quadratic constraint and c is the linear part.

17.1.11 Basis status

For an LP problem with n variables and m constraints, the constraints are treated as slack variables *internally*, resulting in $n + m$ variables. When solving an LP problem using the simplex method, the simplex method fixes n variables at one of their bounds, and then computes solutions for the other m variables. The m variables with computed solution are called *basic* variables, and the other n variables are called *non-basic* variables. The simplex progress and its final solution can be defined using the basis status of all the variables and constraints.

The basis status supported by COPT are:

- `COPT_BASIS_LOWER`

  The variable is non-basic at its lower bound.

- `COPT_BASIS_BASIC`

  The variable is basic.

- `COPT_BASIS_UPPER`

  The variable is non-basic at its upper bound.

- `COPT_BASIS_SUPERBASIC`

  The variable is non-basic but not any of its bounds.

- `COPT_BASIS_FIXED`

  The variable is non-basic and fixed at its bound.

17.1.12 LP solution status

The solution status of an LP problem is called LP status, which can be obtained using integer attribute `COPT_INTATTR_LPSTATUS`.

Possible LP status values are:

- `COPT_LPSTATUS_UNSTARTED`

  The LP optimization is not started yet.

- `COPT_LPSTATUS_OPTIMAL`

  The LP problem is solved to optimality.

- `COPT_LPSTATUS_INFEASIBLE`

  The LP problem is infeasible.

- `COPT_LPSTATUS_UNBOUNDED`

  The LP problem is unbounded.

- `COPT_LPSTATUS_NUMERICAL`

Numerical trouble encountered.

- `COPT_LPSTATUS_TIMEOUT`

  The LP optimization is stopped because of time limit.

- `COPT_LPSTATUS_UNFINISHED`

  The LP optimization is stopped but the solver cannot provide a solution because of numerical difficulties.

- `COPT_LPSTATUS_IMPRECISE`

  The solution is imprecise.

- `COPT_LPSTATUS_INTERRUPTED`

  The LP optimization is stopped by user interrupt.

17.1.13 MIP solution status

The solution status of an MIP problem is called MIP status, which can be obtained using integer attribute `COPT_INTATTR_MIPSTATUS`.

Possible MIP status values are:

- `COPT_MIPSTATUS_UNSTARTED`

  The MIP optimization is not started yet.

- `COPT_MIPSTATUS_OPTIMAL`

  The MIP problem is solved to optimality.

- `COPT_MIPSTATUS_INFEASIBLE`

  The MIP problem is infeasible.

- `COPT_MIPSTATUS_UNBOUNDED`

  The MIP problem is unbounded.

- `COPT_MIPSTATUS_INF_OR_UNB`

  The MIP problem is infeasible or unbounded.

- `COPT_MIPSTATUS_NODELIMIT`

  The MIP optimization is stopped because of node limit.

- `COPT_MIPSTATUS_TIMEOUT`

  The MIP optimization is stopped because of time limit.

- `COPT_MIPSTATUS_UNFINISHED`

  The MIP optimization is stopped but the solver cannot provide a solution because of numerical difficulties.

- `COPT_MIPSTATUS_INTERRUPTED`

  The MIP optimization is stopped by user interrupt.

17.1.14 Callback context

- `COPT_CBCONTEXT_MIPRELAX`

  Invokes the callback when a new LP-relaxation solution is found.

- `COPT_CBCONTEXT_MIPSOL`

  Invokes the callback when a new MIP candidate solution is found.

17.1.15 API function return code

When an API function finishes, it returns an integer **return code**, which indicates whether the API call was finished okay or failed. In case of failure, it specifies the reason of the failure.

Possible COPT API function return codes are:

- `COPT_RETCODE_OK`

  The API call finished successfully.

- `COPT_RETCODE_MEMORY`

  The API call failed because of memory allocation failure.

- `COPT_RETCODE_FILE`

  The API call failed because of file input or output failure.

- `COPT_RETCODE_INVALID`

  The API call failed because of invalid data.

- `COPT_RETCODE_LICENSE`

  The API call failed because of license validation failure. In this case, further information can be obtained by calling `COPT_GetLicenseMsg`.

- `COPT_RETCODE_INTERNAL`

  The API call failed because of internal error.

- `COPT_RETCODE_THREAD`

  The API call failed because of thread error.

- `COPT_RETCODE_SERVER`

  The API call failed because of remote server error.

- `COPT_RETCODE_NONCONVEX`

  The API call failed because of problem is nonconvex.

17.1.16 Client configuration

For floating and cluster clients, users are allowed to set client configuration parameters, currently available settings are:

- `COPT_CLIENT_CLUSTER`

  IP address of cluster server.

- `COPT_CLIENT_FLOATING`

  IP address of token server.

- `COPT_CLIENT_PASSWORD`

  Password of cluster server.

- `COPT_CLIENT_PORT`

  Connection port of token server.

- `COPT_CLIENT_WAITTIME`

  Wait time of client.

17.1.17 Other constants

- `COPT_BUFFSIZE`

  Defines the recommended buffer size when obtaining a C-style string message from COPT library. This can be used with, for example, `COPT_GetBanner`, `COPT_GetRetcodeMsg` etc.

17.2 Attributes

17.2.1 Problem information

- `COPT_INTATTR_COLS` or `"Cols"`

  Integer attribute.

  Number of variables (columns) in the problem.

- `COPT_INTATTR_PSDCOLS` or `"PSDCols"`

  Integer attribute.

  Number of PSD variables in the problem.

- `COPT_INTATTR_ROWS` or `"Rows"`

  Integer attribute.

  Number of constraints (rows) in the problem.

- `COPT_INTATTR_ELEMS` or `"Elems"`

  Integer attribute.

  Number of non-zero elements in the coefficient matrix.

- `COPT_INTATTR_QELEMS` or `"QElems"`

  Integer attribute.

  Number of non-zero quadratic elements in the quadratic objective function.

- `COPT_INTATTR_PSDELEMS` or `"PSDElems"`

  Integer attribute.

  Number of PSD terms in objective function.

- `COPT_INTATTR_SYMMATS` or `"SymMats"`

  Integer attribute.

  Number of symmetric matrices in the problem.

- `COPT_INTATTR_BINS` or `"Bins"`

  Integer attribute.

  Number of binary variables.

- `COPT_INTATTR_INTS` or `"Ints"`

Integer attribute.

Number of integer variables.

- `COPT_INTATTR_SOSS` or `"Soss"`

  Integer attribute.

  Number of SOS constraints.

- `COPT_INTATTR_CONES` or `"Cones"`

  Integer attribute.

  Number of Second-Order-Cone constraints.

- `COPT_INTATTR_QCONSTRS` or `"QConstrs"`

  Integer attribute.

  Number of quadratic constraints.

- `COPT_INTATTR_PSDCONSTRS` or `"PSDConstrs"`

  Integer attribute.

  Number of PSD constraints.

- `COPT_INTATTR_INDICATORS` or `"Indicators"`

  Integer attribute.

  Number of indicator constraints.

- `COPT_INTATTR_OBJSENSE` or `"ObjSense"`

  Integer attribute.

  The optimization direction.

- `COPT_DBLATTR_OBJCONST` or `"ObjConst"`

  Double attribute.

  The constant part of the objective function.

- `COPT_INTATTR_HASQOBJ` or `"HasQObj"`

  Integer attribute.

  Whether the problem has quadratic objective function.

- `COPT_INTATTR_HASPSDOBJ` or `"HasPSDObj"`

  Integer attribute.

  Whether the problem has PSD terms in objective function.

- `COPT_INTATTR_ISMIP` or `"IsMIP"`

  Integer attribute.

  Whether the problem is a MIP.

17.2.2 Solution information

- `COPT_INTATTR_LPSTATUS` or `"LpStatus"`

  Integer attribute.

  The LP status. Please refer to *Constants: LP solution status* for possible values.

- `COPT_INTATTR_MIPSTATUS` or `"MipStatus"`

  Integer attribute.

  The MIP status. Please refer to *Constants: MIP solution status* for possible values.

- `COPT_INTATTR_SIMPLEXITER` or `"SimplexIter"`

  Integer attribute.

  Number of simplex iterations performed.

- `COPT_INTATTR_BARRIERITER` or `"BarrierIter"`

  Integer attribute.

  Number of barrier iterations performed.

- `COPT_INTATTR_NODECNT` or `"NodeCnt"`

  Integer attribute.

  Number of explored nodes.

- `COPT_INTATTR_POOLSOLS` or `"PoolSols"`

  Integer attribute.

  Number of solutions in solution pool.

- `COPT_INTATTR_TUNERESULTS` or `"TuneResults"`

  Integer attribute.

  Number of parameter tuning results

- `COPT_INTATTR_HASLPSOL` or `"HasLpSol"`

  Integer attribute.

  Whether LP solution is available.

- `COPT_INTATTR_HASBASIS` or `"HasBasis"`

  Integer attribute.

  Whether LP basis is available.

- `COPT_INTATTR_HASDUALFARKAS` or `"HasDualFarkas"`

  Integer attribute.

  Whether the dual Farkas of an infeasible LP problem is available.

- `COPT_INTATTR_HASPRIMALRAY` or `"HasPrimalRay"`

  Integer attribute.

  Whether the primal ray of an unbounded LP problem is available.

- `COPT_INTATTR_HASMIPSOL` or `"HasMipSol"`

  Integer attribute.

  Whether MIP solution is available.

- `COPT_INTATTR_IISCOLS` or `"IISCols"`

Integer attribute.

Number of bounds of columns in IIS.

- `COPT_INTATTR_IISROWS` or `"IISRows"`

  Integer attribute.

  Number of rows in IIS.

- `COPT_INTATTR_IISSOSS` or `"IISSOSs"`

  Integer attribute.

  Number of SOS constraints in IIS.

- `COPT_INTATTR_IISINDICATORS` or `"IISIndicators"`

  Integer attribute.

  Number of indicator constraints in IIS.

- `COPT_INTATTR_HASIIS` or `"HasIIS"`

  Integer attribute.

  Whether IIS is available.

- `COPT_INTATTR_HASFEASRELAXSOL` or `"HasFeasRelaxSol"`

  Integer attribute.

  Whether feasibility LP-relaxation solution is available.

- `COPT_INTATTR_ISMINIIS` or `"IsMinIIS"`

  Integer attribute.

  Whether the computed IIS is minimal.

- `COPT_DBLATTR_LPOBJVAL` or `"LpObjval"`

  Double attribute.

  The LP objective value.

- `COPT_DBLATTR_BESTOBJ` or `"BestObj"`

  Double attribute.

  Best integer objective value for MIP.

- `COPT_DBLATTR_BESTBND` or `"BestBnd"`

  Double attribute.

  Best bound for MIP.

- `COPT_DBLATTR_BESTGAP` or `"BestGap"`

  Double attribute.

  Best relative gap for MIP.

- `COPT_DBLATTR_FEASRELAXOBJ` or `FeasRelaxObj`

  Double attribute.

  Feasibility relaxation objective value.

- `COPT_DBLATTR_SOLVINGTIME` or `"SolvingTime"`

  Double attribute.

  The time spent for the optimization (in seconds).

17.3 Information

17.3.1 Problem information

- `COPT_DBLINFO_OBJ` or `"Obj"`

  Double information.

  Objective cost of columns.

- `COPT_DBLINFO_LB` or `"LB"`

  Double information.

  Lower bounds of columns or rows.

- `COPT_DBLINFO_UB` or `"UB"`

  Double information.

  Upper bounds of columns or rows.

17.3.2 Solution information

- `COPT_DBLINFO_VALUE` or `"Value"`

  Double information.

  Solution of columns.

- `COPT_DBLINFO_SLACK` or `"Slack"`

  Double information.

  Solution of slack variables, also known as activities of constraints. Only available for LP problem.

- `COPT_DBLINFO_DUAL` or `"Dual"`

  Double information.

  Solution of dual variables. Only available for LP problem.

- `COPT_DBLINFO_REDCOST` or `"RedCost"`

  Double information.

  Reduced cost of columns. Only available for LP problem.

17.3.3 Dual Farkas and primal ray

Advanced topic. When an LP is infeasible or unbounded, the solver can return the dual Farkas or primal ray to prove it.

- `COPT_DBLINFO_DUALFARKAS` or `"DualFarkas"`

  Double information.

  The dual Farkas for constraints of an infeasible LP problem. Please enable the parameter `"ReqFarkasRay"` to ensure that the dual Farkas is available when the LP is infeasible.

  Without loss of generality, the concept of the dual Farkas can be conveniently demonstrated using an LP problem with general variable bounds and equality constraints: $Ax = 0$ and $l \leq x \leq u$. When the LP is infeasible, a dual Farkas vector y can prove that the system has conflict that $\max y^T Ax < y^T b = 0$. Computing $\max y^T Ax$: with the vector $\hat{a} = y^T A$, choosing variable bound $x_i = l_i$ when $\hat{a}_i < 0$ and $x_i = u_i$ when $\hat{a}_i > 0$ gives the maximal possible value of $y^T Ax$ for any x within their bounds.

Some application relies on the alternate conflict $\min \bar{y}^T Ax > \bar{y}^T b = 0$. This can be achieved by negating the dual Farkas, i.e. $\bar{y} = -y$ returned by the solver.

In very rare cases, the solver may fail to return a valid dual Farkas. For example when the LP problem slightly infeasible by tiny amount, which We recommend to study and to repair the infeasibility using FeasRelax instead.

- `COPT_DBLINFO_PRIMALRAY` or `"PrimalRay"`

  Double information.

  The primal ray for variables of an unbounded LP problem. Please enable the parameter `"ReqFarkasRay"` to ensure that the primal ray is available when an LP is unbounded.

  For a minimization LP problem in the standard form: $\min c^T x, Ax = b$ and $x \geq 0$, a primal ray vector r satisfies that $r \geq 0, Ar = 0$ and $c^T r < 0$.

17.3.4 Feasibility relaxation information

- `COPT_DBLINFO_RELAXLB` or `"RelaxLB"`

  Double information.

  Feasibility relaxation values for lower bounds of columns or rows.

- `COPT_DBLINFO_RELAXUB` or `"RelaxUB"`

  Double information.

  Feasibility relaxation values for upper bounds of columns or rows.

17.4 Callback information

- `COPT_CBINFO_BESTOBJ` or `"BestObj"`

  Double information.

  Current best objective.

- `COPT_CBINFO_BESTBND` or `"BestBnd"`

  Double information.

  Current best objective bound.

- `COPT_CBINFO_INCUMBENT` or `"HasIncumbent"`

  Integer information.

  Whether an incumbent is available.

- `COPT_CBINFO_INCUMBENT` or `"Incumbent"`

  Double information.

  Current best feasible solution.

- `COPT_CBINFO_MIPCANDIDATE` or `"MipCandidate"`

  Double information.

  Current feasible solution candidate.

- `COPT_CBINFO_MIPCANDOBJ` or `"MipCandObj"`

  Double information.

  Objective value for current feasible solution candidate.

- `COPT_CBINFO_RELAXSOLUTION` or `"RelaxSolution"`

Double information.

Current solution of LP-relaxation.

- `COPT_CBINFO_RELAXSOLOBJ` or `"RelaxSolObj"`

  Double information.

  Current objective of LP-relaxation.

17.5 Parameters

17.5.1 Limits and tolerances

- `COPT_DBLPARAM_TIMELIMIT` or `"TimeLimit"`

  Double parameter.

  Time limit of the optimization.

  Default: 1e20

  Minimal: 0

  Maximal: 1e20

- `COPT_INTPARAM_NODELIMIT` or `"NodeLimit"`

  Integer parameter.

  Node limit of the optimization.

  Default: -1

  Minimal: -1

  Maximal: `INT_MAX`

- `COPT_INTPARAM_BARITERLIMIT` or `"BarIterLimit"`

  Integer parameter.

  Iteration limit of barrier method.

  Default: 500

  Minimal: 0

  Maximal: `INT_MAX`

- `COPT_DBLPARAM_MATRIXTOL` or `"MatrixTol"`

  Double parameter.

  Input matrix coefficient tolerance.

  Default: 1e-10

  Minimal: 0

  Maximal: 1e-7

- `COPT_DBLPARAM_FEASTOL` or `"FeasTol"`

  Double parameter.

  The feasibility tolerance.

  Default: 1e-6

  Minimal: 1e-9.

  Maximal: 1e-4

- `COPT_DBLPARAM_DUALTOL` or `"DualTol"`

  Double parameter.

  The tolerance for dual solutions and reduced cost.

  Default: 1e-6

  Minimal: 1e-9

  Maximal: 1e-4

- `COPT_DBLPARAM_INTTOL` or `"IntTol"`

  Double parameter.

  The integrality tolerance for variables.

  Default: 1e-6

  Minimal: 1e-9

  Maximal: 1e-1

- `COPT_DBLPARAM_RELGAP` or `"RelGap"`

  Double parameter.

  The relative gap of optimization.

  Default: 1e-4

  Minimal: 0

  Maximal: `DBL_MAX`

- `COPT_DBLPARAM_ABSGAP` or `"AbsGap"`

  Double parameter.

  The absolute gap of optimization.

  Default: 1e-6

  Minimal: 0

  Maximal: `DBL_MAX`

17.5.2 Presolving and scaling

- `COPT_INTPARAM_PRESOLVE` or `"Presolve"`

  Integer parameter.

  Level of presolving before solving a model.

  Default: -1

  Possible values:

  -1: Automatic

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_SCALING` or `"Scaling"`

Integer parameter.

Whether to perform scaling before solving a problem.

Default: -1

Possible values:

-1: Choose automatically.

0: No scaling.

1: Apply scaling.

- `COPT_INTPARAM_DUALIZE` or `"Dualize"`

  Integer parameter.

  Whether to dualize a problem before solving it.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: No dualizing.

  1: Dualizing the problem.

17.5.3 Linear programming related

- `COPT_INTPARAM_LPMETHOD` or `"LpMethod"`

  Integer parameter.

  Method to solve the LP problem.

  Default: -1

  Possible values:

  -1: Default (Use dual simplex method).

  1: Dual simplex.

  2: Barrier.

  3: Crossover.

  4: Concurrent (Use simplex and barrier simultaneously).

  5: Choose between simplex and barrier automatically (Based on features such as sparsity and/or coefficients ranges).

- `COPT_INTPARAM_DUALPRICE` or `"DualPrice"`

  Integer parameter.

  Specifies the dual simplex pricing algorithm.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Using Devex pricing algorithm.

  1: Using dual steepest-edge pricing algorithm.

- `COPT_INTPARAM_DUALPERTURB` or `"DualPerturb"`

Integer parameter.

Whether to allow the objective function perturbation when using the dual simplex method.

Default: -1

Possible values:

-1: Choose automatically.

0: No perturbation.

1: Allow objective function perturbation.

- `COPT_INTPARAM_BARHOMOGENEOUS` or `"BarHomogeneous"`

  Integer parameter.

  Whether to use homogeneous self-dual form in barrier.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: No.

  1: Yes.

- `COPT_INTPARAM_BARORDER` or `"BarOrder"`

  Integer parameter.

  Barrier ordering algorithm.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Approximate Minimum Degree (AMD).

  1: Nested Dissection (ND).

- `COPT_INTPARAM_CROSSOVER` or `"Crossover"`

  Integer parameter.

  Whether to use crossover.

  Default: 1

  Possible values:

  -1: Choose automatically.

  Only run crossover when the LP solution is not primal-dual feasible.

  0: No.

  1: Yes.

- `COPT_INTPARAM_REQFARKASRAY` or `"ReqFarkasRay"`

  Integer parameter.

  Advanced topic. Whether to compute the dual Farkas or primal ray when the LP is infeasible or unbounded.

  Default: 0

  Possible values:

0: No.

1: Yes.

17.5.4 Semidefinite programming related

- `COPT_INTPARAM_SDPMETHOD` or `"SDPMethod"`

  Integer parameter.

  Method to solve the SDP problem.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Primal-Dual method.

  1: Alternating direction method of multipliers (ADMM).

  2: Dual method.

17.5.5 Integer programming related

- `COPT_INTPARAM_CUTLEVEL` or `"CutLevel"`

  Integer parameter.

  Level of cutting-planes generation.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_ROOTCUTLEVEL` or `"RootCutLevel"`

  Integer parameter.

  Level of cutting-planes generation of root node.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_TREECUTLEVEL` or `"TreeCutLevel"`

Integer parameter.

Level of cutting-planes generation of search tree.

Default: -1

Possible values:

-1: Choose automatically.

0: Off

1: Fast

2: Normal

3: Aggressive

- `COPT_INTPARAM_ROOTCUTROUNDS` or `"RootCutRounds"`

  Integer parameter.

  Rounds of cutting-planes generation of root node.

  Default: -1 (Choose automatically)

  Minimal: -1

  Maximal: `INT_MAX`

- `COPT_INTPARAM_NODECUTROUNDS` or `"NodeCutRounds"`

  Integer parameter.

  Rounds of cutting-planes generation of search tree node.

  Default: -1 (Choose automatically)

  Minimal: -1

  Maximal: `INT_MAX`

- `COPT_INTPARAM_HEURLEVEL` or `"HeurLevel"`

  Integer parameter.

  Level of heuristics.

  Default: -1

  Possible values:

  -1: Choose automatically

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_ROUNDINGHEURLEVEL` or `"RoundingHeurLevel"`

  Integer parameter.

  Level of rounding heuristics.

  Default: -1

  Possible values:

  -1: Choose automatically

  0: Off

  1: Fast

2: Normal

3: Aggressive

- `COPT_INTPARAM_DIVINGHEURLEVEL` or `"DivingHeurLevel"`

  Integer parameter.

  Level of diving heuristics.

  Default: -1

  Possible values:

  -1: Choose automatically

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_TREECUTLEVEL` or `"SubMipHeurLevel"`

  Integer parameter.

  Level of Sub-MIP heuristics.

  Default: -1

  Possible values:

  -1: Choose automatically

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_STRONGBRANCHING` or `"StrongBranching"`

  Integer parameter.

  Level of strong branching.

  Default: -1

  Possible values:

  -1: Choose automatically

  0: Off

  1: Fast

  2: Normal

  3: Aggressive

- `COPT_INTPARAM_CONFLICTANALYSIS` or `"ConflictAnalysis"`

  Integer parameter.

  Whether to perform conflict analysis.

  Default: -1

  Possible values:

-1: Choose automatically.

0: No

1: Yes

- `COPT_INTPARAM_MIPSTARTMODE` or `"MipStartMode"`

  Integer parameter.

  Mode of MIP starts.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Do not use any MIP starts.

  1: Only load full and feasible MIP starts.

  2: Only load feasible ones (complete partial solutions by solving subMIPs).

- `COPT_INTPARAM_MIPSTARTNODELIMIT` or `"MipStartNodeLimit"`

  Integer parameter.

  Limit of nodes for MIP start sub-MIPs.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

  Maximal: `INT_MAX`

17.5.6 Parallel computing related

- `COPT_INTPARAM_THREADS` or `"Threads"`

  Integer parameter.

  Number of threads to use.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

  Maximal: 128

- `COPT_INTPARAM_BARTHREADS` or `"BarThreads"`

  Integer parameter.

  Number of threads used by barrier. If value is -1, the thread count is determined by parameter `Threads`.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

  Maximal: 128

- `COPT_INTPARAM_SIMPLEXTHREADS` or `"SimplexThreads"`

  Integer parameter.

  Number of threads used by dual simplex. If value is -1, the thread count is determined by parameter `Threads`.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

Maximal: 128

- `COPT_INTPARAM_CROSSOVERTHREADS` or `"CrossoverThreads"`

  Integer parameter.

  Number of threads used by crossover. If value is -1, the thread count is determined by parameter `Threads`.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

  Maximal: 128

- `COPT_INTPARAM_MIPTASKS` or `"MipTasks"`

  Integer parameter.

  Number of MIP tasks in parallel.

  Default: -1 (Choose automatically)

  Minimal: -1 (Choose automatically)

  Maximal: 256

17.5.7 IIS computation related

- `COPT_INTPARAM_IISMETHOD` or `"IISMethod"`

  Integer parameter.

  Method for IIS computation.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Find smaller IIS.

  1: Find IIS quickly.

17.5.8 Feasibility relaxation related

- `COPT_INTPARAM_FEASRELAXMODE` or `"FeasRelaxMode"`

  Integer parameter.

  Method for feasibility relaxation.

  Default: 0

  Possible values:

  0: Minimize sum of violations.

  1: Optimize original objective function under minimal sum of violations.

  2: Minimize number of violations.

  3: Optimize original objective function under minimal number of violations.

  4: Minimize sum of squared violations.

  5: Optimize original objective function under minimal sum of squared violations.

17.5.9 Tuner related

- `COPT_DBLPARAM_TUNETIMELIMIT` or `"TuneTimeLimit"`

  Double parameter.

  Time limit for parameter tuning. A value of 0 indicates that the solver is set automatically.

  Default: 0

  Minimal: 0

  Maximal: 1e20

- `COPT_DBLPARAM_TUNETARGETTIME` or `"TuneTargetTime"`

  Double parameter.

  Time target for parameter tuning.

  Default: 1e-2

  Minimal: 0

  Maximal: `DBL_MAX`

- `COPT_DBLPARAM_TUNETARGETRELGAP` or `"TuneTargetRelGap"`

  Double parameter.

  Optimal relative tolerance target for parameter tuning.

  Default: 1e-4

  Minimal: 0

  Maximal: `DBL_MAX`

- `COPT_INTPARAM_TUNEMETHOD` or `"TuneMethod"`

  Integer parameter.

  Method for parameter tuning.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Greedy search strategy.

  1: Broader search strategy.

- `COPT_INTPARAM_TUNEMODE` or `"TuneMode"`

  Integer parameter.

  Mode for parameter tuning.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: Solving time.

  1: Optimal relative tolerance.

  2: Objective function value.

  3: The lower bound of the objective function value.

- `COPT_INTPARAM_TUNEMEASURE` or `"TuneMeasure"`

Integer parameter.

Parameter tuning result calculation method.

Default: -1

Possible values:

-1: Choose automatically.

0: Calculate the average value.

1: Calculate the maximum value.

- `COPT_INTPARAM_TUNEPERMUTES` or `"TunePermutes"`

  Integer parameter.

  Permutations for each trial parameter set. A value of 0 indicates that the solver is automatically set.

  Default: 0

  Minimal: 0

  Maximal: `INT_MAX`

- `COPT_INTPARAM_TUNEOUTPUTLEVEL` or `"TuneOutputLevel"`

  Integer parameter.

  Parameter tuning log output intensity.

  Default: 2

  Possible values:

  0: Not displayed.

  1: Display only improved parameter results.

  2: Displays a summary of the results for each set of parameters.

  3: Display each group of parameter results in detail.

17.5.10 Callback related

- `COPT_INTPARAM_LAZYCONSTRAINTS` or `"LazyConstraints"`

  Integer parameter.

  Whether lazy constraints are part of the model.

  Default: -1

  Possible values:

  -1: Choose automatically.

  0: No.

  1: Yes.

Notes

- This parameter only affects MIP.

17.5.11 Other parameters

- `COPT_INTPARAM_LOGGING` or `"Logging"`

  Integer parameter.

  Whether to print optimization logs.

  Default: 1

  Possible values:

  0: No optimization logs.

  1: Print optimization logs.

- `COPT_INTPARAM_LOGTOCONSOLE` or `"LogToConsole"`

  Integer parameter.

  Whether to print optimization logs to console.

  Default: 1

  Possible values:

  0: No optimization logs to console.

  1: Print optimization logs to console.

17.6 API Functions

The documentations for COPT API functions are grouped by their purposes.

All the return values of COPT API functions are integers, and possible return values are documented in the constants section.

17.6.1 Creating the environment and problem

COPT_CreateEnvConfig

Synopsis

```
int COPT_CreateEnvConfig(copt_env_config **p_config)
```

Description

Create a COPT client configuration.

Arguments

`p_config`

Output pointer to COPT client configuration.

COPT_DeleteEnvConfig

Synopsis

```
int COPT_DeleteEnvConfig(copt_env_config **p_config)
```

Description

Delete COPT client configuration.

Arguments

`p_config`

Input pointer to COPT client configuration.

COPT_SetEnvConfig

Synopsis

```
int COPT_SetEnvConfig(copt_env_config *config, const char *name,
const char *value)
```

Description

Set COPT client configuration.

Arguments

`config`

COPT client configuration.

`name`

Name of configuration parameter.

`value`

Value of configuration parameter.

COPT_CreateEnv

Synopsis

```
int COPT_CreateEnv(copt_env **p_env)
```

Description

Creates a COPT environment.

Calling this function is the first step when using the COPT library. It validates the license, and when the license is okay, the resulting environment variable will allow for creating COPT problems. When the license validation fails, more information can be obtained using `COPT_GetLicenseMsg` to help identify the issue.

Arguments

`p_env`

The output pointer to a variable holding COPT environment.

COPT_CreateEnvWithPath

Synopsis

```
int COPT_CreateEnvWithPath(const char *licDir, copt_env **p_env)
```

Description

Creates a COPT environment, directory of license files is specified by argument `licDir`.

Calling this function is the first step when using the COPT library. It validates the license, and when the license is okay, the resulting environment variable will allow for creating COPT problems. When the license validation fails, more information can be obtained using `COPT_GetLicenseMsg` to help identify the issue.

Arguments

`licDir`

Directory of license files.

`p_env`

Output pointer to a variable holding COPT environment.

COPT_CreateEnvWithConfig

Synopsis

```
int COPT_CreateEnvWithConfig(copt_env_config *config, copt_env
**p_env)
```

Description

Creates a COPT environment, client configuration is specified by argument `config`.

Calling this function is the first step when using the COPT library. It validates the client configuration, and when the license is okay, the resulting environment variable will allow for creating COPT problems. When the license validation fails, more information can be obtained using `COPT_GetLicenseMsg` to help identify the issue.

Arguments

`config`

Client configuration.

`p_env`

Output pointer to a variable holding COPT environment.

COPT_DeleteEnv

Synopsis

```
int COPT_DeleteEnv(copt_env **p_env)
```

Description

Deletes the COPT environment created by `COPT_CreateEnv`.

Arguments

`p_env`

Input pointer to a variable holding COPT environment.

COPT_GetLicenseMsg

Synopsis

```
int COPT_GetLicenseMsg(copt_env *env, char *buff, int buffSize)
```

Description

Returns a C-style string regarding the license validation information. Please refer to this function when `COPT_CreateEnv` fails.

Arguments

`env`

The COPT environment.

`buff`

A buffer for holding the resulting string.

`buffSize`

The size of the above buffer.

COPT_CreateProb

Synopsis

```
int COPT_CreateProb(copt_env *env, copt_prob **p_prob)
```

Description

Creates an empty COPT problem.

Arguments

`env`

The COPT environment.

`p_prob`

Output pointer to a variable holding the COPT problem.

COPT_CreateCopy

Synopsis

```
int COPT_CreateCopy(copt_prob *src_prob, copt_prob **p_dst_prob)
```

Description

Create a deep-copy of an COPT problem.

Note: The parameter settings will be copied too. To solve the copied problem with different parameters, users should reset its parameters to default by calling `COPT_ResetParam` first, and then set parameters as needed.

Arguments

`src_prob`

The pointer to a varialbe hoding the COPT problem to be copied.

`p_dst_prob`

Output pointer to a variable hodling the copied COPT problem.

COPT_DeleteProb

Synopsis

```
int COPT_DeleteProb(copt_prob **p_prob)
```

Description

Deletes the COPT problem created using `COPT_CreateProb`

Arguments

`p_prob`

Input pointer to a variable holding the COPT problem.

17.6.2 Building and modifying a problem

COPT_LoadProb

Synopsis

```
int COPT_LoadProb(
    copt_prob *prob,
    int nCol,
    int nRow,
    int iObjSense,
    double dObjConst,
    const double *obj,
    const int *colMatBeg,
    const int *colMatCnt,
    const int *colMatIdx,
    const double *colMatElem,
    const char *colType,
    const double *colLower,
    const double *colUpper,
    const char *rowSense,
    const double *rowBound,
    const double *rowUpper,
    char const *const *colNames,
    char const *const *rowNames)
```

Description

Loads a problem defined by arrays.

Arguments

`prob`

The COPT problem.

`nCol`

Number of variables (coefficient matrix columns).

`nRow`

Number of constraints (coefficient matrix rows).

`iObjSense`

The optimization sense, either `COPT_MAXIMIZE` or `COPT_MINIMIZE`.

`dObjConst`

The constant part of the objective function.

`obj`

Objective coefficients of variables.

`colMatBeg, colMatCnt, colMatIdx` and `colMatElem`

Defines the coefficient matrix in compressed column storage (CCS) format. Please see **other information** for an example of the CCS format.

If `colMatCnt` is `NULL`, `colMatBeg` will need to have length of `nCol+1`, and the begin and end pointers to the i-th matrix column coefficients are defined using `colMatBeg[i]` and `colMatBeg[i+1]`.

If `colMatCnt` is provided, the begin and end pointers to the i-th column coefficients are defined using `colMatBeg[i]` and `colMatBeg[i] + colMatCnt[i]`.

`colType`

Types of variables.

If `colType` is `NULL`, all variables will be continuous.

`colLower` and `colUpper`

Lower and upper bounds of variables.

If `colLower` is `NULL`, lower bounds will be 0.

If `colUpper` is `NULL`, upper bounds will be infinity.

`rowSense`

Senses of constraint.

Please refer to the list of all senses constants for all the supported types.

If `rowSense` is `NULL`, then `rowBound` and `rowUpper` will be treated as lower and upper bounds for constraints. This is the recommended method for defining constraints.

If `rowSense` is provided, then `rowBound` and `rowUpper` will be treated as RHS and **range** for constraints. In this case, `rowUpper` is only required when there are `COPT_RANGE` constraints, where the

lower bound is `rowBound[i] - fabs(rowUpper[i])`

upper bound is `rowBound[i]`

`rowBound`

Lower bounds or RHS of constraints.

`rowUpper`

Upper bounds or **range** of constraints.

`colNames` and `rowNames`

Names of variables and constraints. Can be `NULL`.

Other information

The compressed column storage (CCS) is a common format for storing sparse matrix. We demonstrate how to store the example matrix with 4 columns and 3 rows in the CCS format.

$$A = \begin{bmatrix} 1.1 & 1.2 & & \\ & 2.2 & 2.3 & \\ & & 3.3 & 3.4 \end{bmatrix} \tag{17.6}$$

```
// Compressed column storage using colMatBeg
colMatBeg[5]  = {  0,   1,   3,   5,   6};
colMatIdx[6]  = {  0,   0,   1,   1,   2,   2};
colMatElem[6] = {1.1, 1.2, 2.2, 2.3, 3.3, 3.4};

// Compressed column storage using both colMatBeg and colMatCnt.
// The * in the example represents unused spaces.
colMatBeg[4]  = {  0,   1,   5,   7};
colMatCnt[4]  = {  1,   2,   2,   1};
colMatIdx[6]  = {  0,   0,   1,   1,   2,   *,   *,   2};
colMatElem[6] = {1.1, 1.2, 2.2, 2.3, 3.3,   *,   *, 3.4};
```

COPT_AddCol

Synopsis

```
int COPT_AddCol(
    copt_prob *prob,
    double dColObj,
    int nColMatCnt,
    const int *colMatIdx,
    const double *colMatElem,
    char cColType,
    double dColLower,
    double dColUpper,
    const char *colName)
```

Description

Adds one variable (column) to the problem.

Arguments

`prob`

The COPT problem.

`dColObj`

The objective coefficient of the variable.

`nColMatCnt`

Number of non-zero elements in the column.

`colMatIdx`

Row index of non-zero elements in the column.

`colMatElem`

Values of non-zero elements in the column.

`cColType`

The type of the variable.

`dColLower` and `dColUpper`

The lower and upper bounds of the variable.

`colName`

The name of the variable. Can be `NULL`.

COPT_AddPSDCol

Synopsis

```
int COPT_AddPSDCol(copt_prob *prob, int colDim, const char *name)
```

Description

Add a PSD variable to the problem.

Arguments

`prob`

The COPT problem.

`colDim`

Dimension of new PSD variable.

`name`

Name of new PSD variable. Can be `NULL`.

COPT_AddRow

Synopsis

```
int COPT_AddRow(
    copt_prob *prob,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense,
    double dRowBound,
    double dRowUpper,
    const char *rowName)
```

Description

Adds one constraint (row) to the problem.

Arguments

`prob`

The COPT problem.

`nRowMatCnt`

Number of non-zero elements in the row.

`rowMatIdx`

Column index of non-zero elements in the row.

`rowMatElem`

Values of non-zero elements in the row.

`cRowSense`

The sense of the row.

Please refer to the list of all senses constants for all the supported types.

If `cRowSense` is 0, then `dRowBound` and `dRowUpper` will be treated as lower and upper bounds for the constraint. This is the recommended method for defining constraints.

If `cRowSense` is provided, then `dRowBound` and `dRowUpper` will be treated as RHS and **range** for the constraint. In this case, `dRowUpper` is only required when `cRowSense = COPT_RANGE`, where

lower bound is `dRowBound - dRowUpper`

upper bound is `dRowBound`

`dRowBound`

Lower bound or RHS of the constraint.

`dRowUpper`

Upper bound or **range** of the constraint.

`rowName`

The name of the constraint. Can be `NULL`.

COPT_AddCols

Synopsis

```
int COPT_AddCols(
    copt_prob *prob,
    int nAddCol,
    const double *colObj,
    const int *colMatBeg,
    const int *colMatCnt,
    const int *colMatIdx,
    const double *colMatElem,
    const char *colType,
    const double *colLower,
    const double *colUpper,
    char const *const *colNames)
```

Description

Adds `nAddCol` variables (columns) to the problem.

Arguments

`prob`

The COPT problem.

`nAddCol`

Number of new variables.

`colObj`

Objective coefficients of new variables.

`colMatBeg`, `colMatCnt`, `colMatIdx` and `colMatElem`

Defines the coefficient matrix in compressed column storage (CCS) format. Please see **other information** of `COPT_LoadProb` for an example of the CCS format.

`colType`

Types of new variables.

`colLower` and `colUpper`

Lower and upper bounds of new variables.

If `colLower` is `NULL`, lower bounds will be 0.

If `colUpper` is `NULL`, upper bounds will be `COPT_INFINITY`.

`colNames`

Names of new variables. Can be `NULL`.

COPT_AddPSDCols

Synopsis

```
int COPT_AddPSDCols(copt_prob *prob, int nAddCol, const int*
colDims, char const *const *names)
```

Description

Add `nAddCol` PSD variables to the problem.

Arguments

`prob`

The COPT problem.

`nAddCol`

Number of new PSD variables.

`colDims`

Dimensions of new PSD variables.

`names`

Names of new PSD variables. Can be `NULL`.

COPT_AddRows

Synopsis

```
int COPT_AddRows(
    copt_prob *prob,
    int nAddRow,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const char *rowSense,
    const double *rowBound,
    const double *rowUpper,
    char const *const *rowNames)
```

Description

Adds `nAddRow` constraints (rows) to the problem.

Arguments

`prob`

The COPT problem.

`nAddRow`

Number of new constraints.

`rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`rowSense`

Senses of new constraints.

Please refer to the list of all senses constants for all the supported types.

If `rowSense` is `NULL`, then `rowBound` and `rowUpper` will be treated as lower and upper bounds for constraints. This is the recommended method for defining constraints.

If `rowSense` is provided, then `rowBound` and `rowUpper` will be treated as RHS and **range** for constraints. In this case, `rowUpper` is only required when there are `COPT_RANGE` constraints, where the

lower bound is `rowBound[i] - fabs(rowUpper[i])`

upper bound is `rowBound[i]`

`rowBound`

Lower bounds or RHS of new constraints.

`rowUpper`

Upper bounds or **range** of new constraints.

`rowNames`

Names of new constraints. Can be `NULL`.

COPT_AddSOSs

Synopsis

```
int COPT_AddSOSs(
    copt_prob *prob,
    int nAddSOS,
    const int *sosType,
    const int *sosMatBeg,
    const int *sosMatCnt,
    const int *sosMatIdx,
    const double *sosMatWt)
```

Description

Add `nAddSOS` SOS constraints to the problem. If `sosMatWt` is `NULL`, then COPT will generate it internally.

Note: if a problem contains SOS constraints, the problem is a MIP.

Arguments

`prob`

The COPT problem.

`nAddSOS`

Number of new SOS constraints.

`sosType`

Types of SOS constraints.

`sosMatBeg`, `sosMatCnt`, `sosMatIdx` and `sosMatWt`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`sosMatWt`

Weights of variables in SOS constraints. Can be `NULL`.

COPT_AddCones

Synopsis

```
int COPT_AddCones(
    copt_prob *prob,
    int nAddCone,
    const int *coneType,
    const int *coneBeg,
    const int *coneCnt,
    const int *coneIdx)
```

Description

Add `nAddCone` Second-Order-Cone (SOC) constraints.

Arguments

`prob`

The COPT problem.

`nAddCone`

Number of new SOC constraints.

`coneType`

Types of SOC constraints.

`coneBeg, coneCnt, coneIdx`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

COPT_AddQConstr

Synopsis

```
int COPT_AddQConstr(
    copt_prob *prob,
    int nRowMatCnt,
    const int *rowMatIdx,
    const int *rowMatElem,
    int nQMatCnt,
    const int *qMatRow,
    const int *qMatCol,
    const double *qMatElem,
    char cRowSense,
    double dRowBound, const char *name)
```

Description

Add a general quadratic constraint.

Note Only convex quadratic constraint is currently supported.

Arguments

`prob`

The COPT problem.

`nRowMatCnt`

Number of non-zero linear terms of the quadratic constraint (row).

`rowMatIdx`

Column index of non-zero linear terms of the quadratic constraint (row).

`rowMatElem`

Values of non-zero linear terms of the quadratic constraint (row).

`nQMatCnt`

Number of non-zero quadratic terms of the quadratic constraint (row).

`qMatRow`

Row index of non-zero quadratic terms of the quadratic constraint (row).

`qMatCol`

Column index of non-zero quadratic terms of the quadratic constraint (row).

`qMatElem`

Values of non-zero quadratic terms of the quadratic constraint (row).

`cRowSense`

The sense of the quadratic constraint (row).

`dRowBound`

Right hand side of the quadratic constraint (row).

`name`

Name of the quadratic constraint (row).

COPT_AddPSDConstr

Synopsis

```
int COPT_AddPSDConstr(
    copt_prob *prob,
    int nRowMatCnt,
    const int *rowMatIdx,
    const int *rowMatElem,
    int nColCnt,
    const int *psdColIdx,
    const int *symMatIdx,
    char cRowSense,
    double dRowBound,
    double dRowUpper,
    const char *name)
```

Description

Add a PSD constraint.

Arguments

`prob`

The COPT problem.

`nRowMatCnt`

Number of non-zero linear terms of the PSD constraint.

`rowMatIdx`

Column index of non-zero linear terms of the PSD constraint.

`rowMatElem`

Values of non-zero linear terms of the PSD constraint.

`nColCnt`

Number of PSD terms of the PSD constraint.

`psdColIdx`

PSD variable index of PSD terms of the PSD constraint.

`symMatIdx`

Symmetric matrix index of PSD terms of the PSD constraint.

`cRowSense`

Senses of new PSD constraint.

Please refer to the list of all senses constants for all the supported types.

If `cRowSense` is 0, then `dRowBound` and `dRowUpper` will be treated as lower and upper bounds for the constraint. This is the recommended method for defining constraints.

If `cRowSense` is provided, then `dRowBound` and `dRowUpper` will be treated as RHS and **range** for the constraint. In this case, `dRowUpper` is only required when `cRowSense = COPT_RANGE`, where

lower bound is `dRowBound - dRowUpper`

upper bound is `dRowBound`

`dRowBound`

Lower bound or RHS of the PSD constraint.

`dRowUpper`

Upper bound or **range** of the PSD constraint.

`name`

Name of the PSD constraint. Can be `NULL`.

COPT_AddIndicator

Synopsis

```
int COPT_AddIndicator(
    copt_prob *prob,
    int binColIdx,
    int binColVal,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense, double dRowBound)
```

Description

Add an indicator constraint to the problem.

Arguments

`prob`

The COPT problem.

`binColIdx`

Index of indicator variable (column).

`binColVal`

Value of indicator variable (column).

`nRowMatCnt`

Number of non-zero elements in the linear constraint (row).

`rowMatIdx`

Column index of non-zero elements in the linear constraint (row).

`rowMatElem`

Values of non-zero elements in the linear constraint (row).

`cRowSense`

The sense of the linear constraint (row). Options are: `COPT_EQUAL` , `COPT_LESS_EQUAL` and `COPT_GREATER_EQUAL` .

`dRowBound`

Right hand side of the linear constraint (row).

COPT_AddSymMat

Synopsis

```
int COPT_AddSymMat(copt_prob *prob, int ndim, int nelem, int *rows,
int *cols, double *elems)
```

Description

Add a symmetric matrix to the problem. (Expect lower triangle part)

Arguments

`prob`

The COPT problem.

`ndim`

Dimension of symmetric matrix.

`nelem`

Number of non-zeros of symmetric matrix.

`rows`

Row index of symmetric matrix.

`cols`

Column index of symmetric matrix.

`elems`

Nonzero elements of symmetric matrix.

COPT_DelCols

Synopsis

```
int COPT_DelCols(copt_prob *prob, int num, const int *list)
```

Description

Deletes `num` variables (columns) from the problem.

Arguments

`prob`

The COPT problem.

`num`

Number of variables to be deleted.

`list`

A list of index of variables to be deleted.

COPT_DelPSDCols

Synopsis

```
int COPT_DelPSDCols(copt_prob *prob, int num, const int *list)
```

Description

Deletes `num` PSD variables from the problem.

Arguments

`prob`

The COPT problem.

`num`

Number of PSD variables to be deleted.

`list`

A list of index of PSD variables to be deleted.

COPT_DelRows

Synopsis

```
int COPT_DelRows(copt_prob *prob, int num, const int *list)
```

Description

Deletes `num` constraints (rows) from the problem.

Arguments

`prob`

The COPT problem.

`num`

The number of constraints to be deleted.

`list`

The list of index of constraints to be deleted.

COPT_DelSOSs

Synopsis

```
int COPT_DelSOSs(copt_prob *prob, int num, const int *list)
```

Description

Deletes `num` SOS constraints from the problem.

Arguments

`prob`

The COPT problem.

`num`

The number of SOS constraints to be deleted.

`list`

The list of index of SOS constraints to be deleted.

COPT_DelCones

Synopsis

```
int COPT_DelCones(copt_prob *prob, int num, const int *list)
```

Description

Deletes `num` Second-Order-Cone (SOC) constraints from the problem.

Arguments

`prob`

The COPT problem.

`num`

The number of SOC constraints to be deleted.

`list`

The list of index of SOC constraints to be deleted.

COPT_DelQConstrs

Synopsis

```
int COPT_DelQConstrs(copt_prob *prob, int num, const int *list)
```

Description

Deletes `num` quadratic constraints from the problem.

Arguments

`prob`

The COPT problem.

`num`

The number of quadratic constraints to be deleted.

`list`

The list of index of quadratic constraints to be deleted.

COPT_DelPSDConstrs

Synopsis

```
int COPT_DelPSDConstrs(copt_prob *prob, int num, const int *list)
```

Description

Deletes `num` PSD constraints from the problem.

Arguments

`prob`

The COPT problem.

`num`

The number of PSD constraints to be deleted.

`list`

The list of index of PSD constraints to be deleted.

COPT_DelIndicators

Synopsis

```
int COPT_DelIndicators(copt_prob *prob, int num, const int *list)
```

Description

Deletes `num` indicator constraints from the problem.

Arguments

`prob`

The COPT problem.

`num`

The number of indicator constraints to be deleted.

`list`

The list of index of indicator constraints to be deleted.

COPT_DelQuadObj

Synopsis

```
int COPT_DelQuadObj(copt_prob *prob)
```

Description

Deletes the quadratic terms from the quadratic objective function.

Arguments

`prob`

The COPT problem.

COPT_DelPSDObj

Synopsis

```
int COPT_DelPSDObj(copt_prob *prob)
```

Description

Deletes the PSD terms from objective function.

Arguments

`prob`

The COPT problem.

COPT_SetElem

Synopsis

```
int COPT_SetElem(copt_prob *prob, int iCol, int iRow, double
newElem)
```

Description

Set coefficient of specified row and column.

Note: If `newElem` is smaller than or equal to parameter `MatrixTol`, the coefficient will be set as zero.

Arguments

`prob`

The COPT problem.

`iCol`

Column index.

`iRow`

Row index.

`newElem`

New coefficient.

COPT_SetPSDElem

Synopsis

```
int COPT_SetPSDElem(copt_prob *prob, int iCol, int iRow, int newIdx)
```

Description

Set symmetric matrix index for given PSD term of PSD constraint.

Arguments

`prob`

The COPT problem.

`iCol`

PSD variable index.

`iRow`

PSD constraint index.

`newIdx`

New symmetric matrix index.

COPT_SetObjSense

Synopsis

```
int COPT_SetObjSense(copt_prob *prob, int iObjSense)
```

Description

Change the objective function sense.

Arguments

`prob`

The COPT problem.

`iObjSense`

The optimization sense, either `COPT_MAXIMIZE` or `COPT_MINIMIZE`.

COPT_SetObjConst

Synopsis

```
int COPT_SetObjConst(copt_prob *prob, double dObjConst)
```

Description

Set the constant term of objective function.

Arguments

`prob`

The COPT problem.

`dObjConst`

The constant term of objective function.

COPT_SetColObj/Type/Lower/Upper/Names

Synopsis

```
int COPT_SetColObj(copt_prob *prob, int num, const int *list, const
double *obj)

int COPT_SetColType(copt_prob *prob, int num, const int *list, const
char *type)

int COPT_SetColLower(copt_prob *prob, int num, const int *list,
const double *lower)

int COPT_SetColUpper(copt_prob *prob, int num, const int *list,
const double *upper)

int COPT_SetColNames(copt_prob *prob, int num, const int *list, char
const *const *names)
```

Description

These five API functions each modifies

objective coefficients

variable types

lower bounds

upper bounds

names

of `num` variables (columns) in the problem.

Arguments

`prob`

The COPT problem.

`num`

Number of variables to modify.

`list`

A list of index of variables to modify.

`obj`

New objective coefficients for each variable in the `list`.

`types`

New types for each variable in the `list`.

`lower`

New lower bounds for each variable in the `list`.

`upper`

New upper bounds for each variable in the `list`.

`names`

New names for each variable in the `list`.

COPT_SetPSDColNames

Synopsis

```
int COPT_SetPSDColNames(copt_prob *prob, int num, const int *list,
char const *const *names)
```

Description

Modify names of `num` PSD variables.

Arguments

`prob`

The COPT problem.

`num`

Number of PSD variables to modify.

`list`

A list of index of PSD variables to modify.

`names`

New names for each PSD variable in the `list`.

COPT_SetRowLower/Upper/Names

Synopsis

```
int COPT_SetRowLower(copt_prob *prob, int num, const int *list,
const double *lower)
```

```
int COPT_SetRowUpper(copt_prob *prob, int num, const int *list,
const double *upper)
```

```
int COPT_SetRowNames(copt_prob *prob, int num, const int *list, char
const *const *names)
```

Description

These three API functions each modifies

lower bounds

upper bounds

names

of `num` constraints (rows) in the problem.

Arguments

`prob`

The COPT problem.

`num`

Number of constraints to modify.

`list`

A list of index of constraints to modify.

`lower`

New lower bounds for each constraint in the `list`.

`upper`

New upper bounds for each constraint in the `list`.

`names`

New names for each constraint in the `list`.

COPT_SetQConstrSense/Rhs/Names

Synopsis

```
int COPT_SetQConstrSense(copt_prob *prob, int num, const int *list,
const char *sense)
```

```
int COPT_SetQConstrRhs(copt_prob *prob, int num, const int *list,
const double *rhs)
```

```
int COPT_SetQConstrNames(copt_prob *prob, int num, const int *list,
char const *const *names)
```

Description

These three API functions each modifies

senses

RHS

names

of `num` quadratic constraints (rows) in the problem.

Arguments

`prob`

The COPT problem.

`num`

Number of quadratic constraints to modify.

`list`

A list of index of quadratic constraints to modify.

`sense`

New senses for each quadratic constraint in the `list`.

`rhs`

New RHS for each quadratic constraint in the `list`.

`names`

New names for each quadratic constraint in the `list`.

COPT_SetPSDConstrLower/Upper/Names

Synopsis

```
int COPT_SetPSDConstrLower(copt_prob *prob, int num, const int
*list, const double *lower)
```

```
int COPT_SetPSDConstrUpper(copt_prob *prob, int num, const int
*list, const double *upper)
```

```
int COPT_SetPSDConstrNames(copt_prob *prob, int num, const int
*list, char const *const *names)
```

Description

These three API functions each modifies

lower bounds

upper bounds

names

of `num` PSD constraints in the problem.

Arguments

`prob`

The COPT problem.

`num`

Number of PSD constraints to modify.

`list`

A list of index of PSD constraints to modify.

`lower`

New lower bounds for each PSD constraint in the `list`.

`upper`

New upper bounds for each PSD constraint in the `list`.

`names`

New names for each PSD constraint in the `list`.

COPT_ReplaceColObj

Synopsis

```
int COPT_ReplaceColObj(copt_prob *prob, int num, const int *list,
const double *obj)
```

Description

Replace objective function with new objective function represented by specified objective costs.

Arguments

`prob`

The COPT problem.

`num`

Number of variables to be modified.

`list`

Index of variables to be modified.

`obj`

Objective costs of modified variables.

COPT_ReplacePSDObj

Synopsis

```
int COPT_ReplacePSDObj(copt_prob *prob, int num, const int *list,
const int *idx)
```

Description

Replace PSD terms in objective function with specified PSD terms.

Arguments

`prob`

The COPT problem.

`num`

Number of PSD variables to be modified.

`list`

Index of PSD variables to be modified.

`idx`

Symmetric matrix index of modified PSD variables.

COPT_SetQuadObj

Synopsis

```
int COPT_SetQuadObj(copt_prob *prob, int num, int *qRow, int *qCol,
double *qElem)
```

Description

Set the quadratic terms of the quadratic objective function.

Arguments

`prob`

The COPT problem.

`num`

Number of non-zero quadratic terms of the quadratic objective function.

`qRow`

Row index of non-zero quadratic terms of the quadratic objective function.

`qCol`

Column index of non-zero quadratic terms of the quadratic objective function.

`qElem`

Values of non-zero quadratic terms of the quadratic objective function.

COPT_SetPSDObj

Synopsis

```
int COPT_SetPSDObj(copt_prob *prob, int iCol, int newIdx)
```

Description

Set PSD terms of objective function.

Arguments

`prob`

The COPT problem.

`iCol`

PSD variable index.

`newIdx`

Symmetric matrix index.

17.6.3 Reading and writing the problem

COPT_ReadMps

Synopsis

```
int COPT_ReadMps(copt_prob *prob, const char *mpsfilename)
```

Description

Reads a problem from a MPS file.

Arguments

`prob`

The COPT problem.

`mpsfilename`

The path to the MPS file.

COPT_ReadLp

Synopsis

```
int COPT_ReadLp(copt_prob *prob, const char *lpfilename)
```

Description

Read a problem from a LP file.

Arguments

`prob`

The COPT problem.

`lpfilename`

The path to the LP file.

COPT_ReadSDPA

Synopsis

```
int COPT_ReadSDPA(copt_prob *prob, const char *sdpafilename)
```

Description

Reads a problem from SDPA format file.

Arguments

`prob`

The COPT problem.

`sdpafilename`

The path to the SDPA format file.

COPT_ReadCbf

Synopsis

```
int COPT_ReadCbf(copt_prob *prob, const char *cbffilename)
```

Description

Reads a problem from CBF format file.

Arguments

`prob`

The COPT problem.

`cbffilename`

The path to the CBF format file.

COPT_ReadBin

Synopsis

```
int COPT_ReadBin(copt_prob *prob, const char *binfilename)
```

Description

Reads a problem from a COPT binary format file.

Arguments

`prob`

The COPT problem.

`binfilename`

The path to the COPT binary format file.

COPT_ReadBlob

Synopsis

```
int COPT_ReadBlob(copt_prob *prob, void *blob, COPT_INT64 len)
```

Description

Reads a problem from COPT serialized data.

Arguments

`prob`

The COPT problem.

`blob`

Serialized data.

`len`

Length of serialized data.

COPT_WriteMps

Synopsis

```
int COPT_WriteMps(copt_prob *prob, const char *mpsfilename)
```

Description

Writes the problem to a MPS file.

Arguments

`prob`

The COPT problem.

`mpsfilename`

The path to the MPS file.

COPT_WriteMpsStr

Synopsis

```
int COPT_WriteMpsStr(copt_prob *prob, char *str, int nStrSize, int
*pReqSize)
```

Description

Writes the problem to a string buffer as MPS format.

Arguments

`prob`

The COPT problem.

`str`

String buffer of MPS-format problem.

`nStrSize`

The size of string buffer.

`pReqSize`

Minimum space requirement of string buffer for problem.

COPT_WriteLp

Synopsis

```
int COPT_WriteLp(copt_prob *prob, const char *lpfilename)
```

Description

Writes the problem to a LP file.

Arguments

`prob`

The COPT problem.

`lpfilename`

The path to the LP file.

COPT_WriteCbf

Synopsis

```
int COPT_WriteCbf(copt_prob *prob, const char *cbffilename)
```

Description

Writes the problem to a CBF format file.

Arguments

`prob`

The COPT problem.

`cbffilename`

The path to the CBF format file.

COPT_WriteBin

Synopsis

```
int COPT_WriteBin(copt_prob *prob, const char *binfilename)
```

Description

Writes the problem to a COPT binary format file.

Arguments

`prob`

The COPT problem.

`binfilename`

The path to the COPT binary format file.

COPT_WriteBlob

Synopsis

```
int COPT_WriteBlob(copt_prob *prob, int tryCompress, void **p_blob,
COPT_INT64 *pLen)
```

Description

Writes the problem to COPT serialized data.

Arguments

`prob`

The COPT problem.

`tryCompress`

Whether to compress data.

`p_blob`

Output pointer of serialized data.

`pLen`

Pointer to length of serialized data.

17.6.4 Solving the problem and accessing solutions

COPT_SolveLp

Synopsis

```
int COPT_SolveLp(copt_prob *prob)
```

Description

Solves the LP, QP, QCP, SOCP and SDP problem.

If problem is a MIP, then integer restrictions on variables will be ignored, and SOS constraints, indicator constraints will be discarded, and the problem will be solved as a LP.

Arguments

`prob`

The COPT problem.

COPT_Solve

Synopsis

```
int COPT_Solve(copt_prob *prob)
```

Description

Solves the problem.

Arguments

`prob`

The COPT problem.

COPT_GetSolution

Synopsis

```
int COPT_GetSolution(copt_prob *prob, double *colVal)
```

Description

Obtains MIP solution.

Arguments

`prob`

The COPT problem.

`colVal`

Solution values of variables.

COPT_GetPoolObjVal

Synopsis

```
int COPT_GetPoolObjVal(copt_prob *prob, int iSol, double *p_objVal)
```

Description

Obtains the `iSol` -th objective value in solution pool.

Arguments

`prob`

The COPT problem.

`iSol`

Index of solution.

`p_objVal`

Pointer to objective value.

COPT_GetPoolSolution

Synopsis

```
int COPT_GetPoolSolution(copt_prob *prob, int iSol, int num, const
int *list, double *colVal)
```

Description

Obtains the `iSol` -th solution.

Arguments

`prob`

The COPT problem.

`iSol`

Index of solution.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`colVal`

Array of solution.

COPT_GetLpSolution

Synopsis

```
int COPT_GetLpSolution(copt_prob *prob, double *value, double
*slack, double *rowDual, double *redCost)
```

Description

Obtains LP, QP, QCP, SOCP and SDP solutions.

Note: For SDP, please use `COPT_GetPSDColInfo` to obtain primal/dual solution of PSD variable.

Arguments

`prob`

The COPT problem.

`value`

Solution values of variables. Can be `NULL`.

`slack`

Solution values of slack variables. They are also known as activities of constraints. Can be `NULL`.

`rowDual`

Dual values of constraints. Can be `NULL`.

`redCost`

Reduced cost of variables. Can be `NULL`.

COPT_SetLpSolution

Synopsis

```
int COPT_SetLpSolution(copt_prob *prob, double *value, double
*slack, double *rowDual, double *redCost)
```

Description

Set LP solution.

Arguments

`prob`

The COPT problem.

`value`

Solution values of variables.

`slack`

Solution values of slack variables.

`rowDual`

Dual values of constraints.

`redCost`

Reduced cost of variables.

COPT_GetBasis

Synopsis

```
int COPT_GetBasis(copt_prob *prob, int *colBasis, int *rowBasis)
```

Description

Obtains LP basis.

Arguments

`prob`

The COPT problem.

`colBasis` and `rowBasis`

The basis status of variables and constraints. Please refer to basis constants for possible values and their meanings.

COPT_SetBasis

Synopsis

```
int COPT_SetBasis(copt_prob *prob, const int *colBasis, const int
*rowBasis)
```

Description

Sets LP basis. It can be used to warm-start an LP optimization.

Arguments

`prob`

The COPT problem.

`colBasis` and `rowBasis`

The basis status of variables and constraints. Please refer to basis constants for possible values and their meanings.

COPT_SetSlackBasis

Synopsis

```
int COPT_SetSlackBasis(copt_prob *prob)
```

Description

Sets a slack basis for LP. The slack basis is the default starting basis for an LP problem. This API function can be used to restore an LP problem to its starting basis.

Arguments

`prob`

The COPT problem.

COPT_Reset

Synopsis

```
int COPT_Reset(copt_prob *prob, int iClearAll)
```

Description

Reset basis and LP/MIP solution in problem, which forces next solve start from scratch. If `iClearAll` is `1`, then clear additional information such as MIP start as well.

Arguments

`prob`

The COPT problem.

`iClearAll`

Whether to clear additional information.

COPT_ReadSol

Synopsis

```
int COPT_ReadSol(copt_prob *prob, const char *solfilename)
```

Description

Reads a MIP solution from file as MIP start information.

Note: The default solution value is 0, i.e. a partial solution will be automatically filled in with zeros.

Arguments

`prob`

The COPT problem.

`solfilename`

The path to the solution file.

COPT_WriteSol

Synopsis

```
int COPT_WriteSol(copt_prob *prob, const char *solfilename)
```

Description

Writes a LP/MIP solution to a file.

Arguments

`prob`

The COPT problem.

`solfilename`

The path to the solution file.

COPT_WritePoolSol

Synopsis

```
int COPT_WritePoolSol(copt_prob *prob, int iSol, const char
*solfilename)
```

Description

Writes selected pool solution to a file.

Arguments

`prob`

The COPT problem.

`iSol`

Index of pool solution.

`solfilename`

The path to the solution file.

COPT_WriteBasis

Synopsis

```
int COPT_WriteBasis(copt_prob *prob, const char *basfilename)
```

Description

Writes the internal LP basis to a file.

Arguments

`prob`

The COPT problem.

`basfilename`

The path to the basis file.

COPT_ReadBasis

Synopsis

```
int COPT_ReadBasis(copt_prob *prob, const char *basfilename)
```

Description

Reads the LP basis from a file. It can be used to warm-start an LP optimization.

Arguments

`prob`

The COPT problem.

`basfilename`

The path to the basis file.

17.6.5 Accessing information of problem

COPT_GetCols

Synopsis

```
int COPT_GetCols(
    copt_prob *prob,
    int nCol,
    const int *list,
    int *colMatBeg,
    int *colMatCnt,
    int *colMatIdx,
    double *colMatElem,
    int nElemSize,
    int *pReqSize)
```

Description

Extract coefficient matrix by columns.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `colMatBeg`, `colMatCnt`, `colMatIdx` and `colMatElem`, we get number of non-zeros elements by `pReqSize` specified by `nCol` and `list`. Secondly, allocate sufficient memory for CCS-format matrix and call this function again to extract coefficient matrix. If the memory of coefficient matrix passed to function is not sufficient, then return the first `nElemSize` non-zero elements, and the minimal required length of non-zero elements by `pReqSize`. If `list` is `NULL`, then the first `nCol` columns will be returned.

Arguments

`prob`

The COPT problem.

`nCol`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`colMatBeg, colMatCnt, colMatIdx` and `colMatElem`

Defines the coefficient matrix in compressed column storage (CCS) format. Please see **other information** of `COPT_LoadProb` for an example of the CCS format.

`nElemSize`

Length of array for non-zero coefficients.

`pReqSize`

Pointer to minimal length of array for non-zero coefficients. Can be `NULL`.

COPT_GetPSDCols

Synopsis

```
int COPT_GetPSDCols(copt_prob *prob, int nCol, int *list, int*
colDims, int *colLens)
```

Description

Get dimension and flattened length of `nCol` PSD variables.

Arguments

`prob`

The COPT problem.

`nCol`

Number of PSD variables.

`list`

Index of PSD variables. Can be `NULL`.

`colDims`

Dimension of PSD variables.

`colLens`

Flattened length of PSD variables.

COPT_GetRows

Synopsis

```
int COPT_GetRows(
    copt_prob *prob,
    int nRow,
    const int *list,
    int *rowMatBeg,
    int *rowMatCnt,
    int *rowMatIdx,
    double *rowMatElem,
    int nElemSize,
    int *pReqSize)
```

Description

Extract coefficient matrix by rows.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`, we get number of non-zeros elements by `pReqSize` specified by `nRow` and `list`. Secondly, allocate sufficient memory for CRS-format matrix and call this function again to extract coefficient matrix. If the memory of coefficient matrix passed to function is not sufficient, then return the first `nElemSize` non-zero elements, and the minimal required length of non-zero elements by `pReqSize`. If `list` is `NULL`, then the first `nCol` columns will be returned.

Arguments

`prob`

The COPT problem.

`nRow`

Number of rows.

`list`

Index of rows. Can be `NULL`.

`rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. Please see **other information** of `COPT_LoadProb` for an example of the CRS format.

`nElemSize`

Length of array for non-zero coefficients.

`pReqSize`

Pointer to minimal length of array for non-zero coefficients. Can be `NULL`.

COPT_GetElem

Synopsis

```
int COPT_GetElem(copt_prob *prob, int iCol, int iRow, double
*p_elem)
```

Description

Get coefficient of specified row and column.

Arguments

`prob`

The COPT problem.

`iCol`

Column index.

`iRow`

Row index.

`p_elem`

Pointer to requested coefficient.

COPT_GetPSDElem

Synopsis

```
int COPT_GetPSDElem(copt_prob *prob, int iCol, int iRow, int *p_idx)
```

Description

Get symmetric matrix index of specified PSD constraint and PSD variable.

Arguments

`prob`

The COPT problem.

`iCol`

PSD variable index.

`iRow`

PSD constraint index.

`p_idx`

Pointer to requested symmetric matrix index.

COPT_GetSymMat

Synopsis

```
int COPT_GetSymMat(
    copt_prob *prob,
    int iMat,
    int *p_nDim,
    int *p_nElem,
    int *rows,
```

```
int *cols,
double *elems)
```

Description

Get specified symmetric matrix.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `rows`, `cols` and `elems`, we get dimension and number of non-zeros of symmetric matrix by `p_nDim` and `p_nElem`, then allocate enough memory for `rows`, `cols` and `elems` and call this function to get the data of symmetric matrix.

Arguments

`prob`

The COPT problem.

`iMat`

Symmetric matrix index.

`p_nDim`

Pointer to dimension of symmetric matrix.

`p_nElem`

Pointer to number of nonzeros of symmetric matrix.

`rows`

Row index of symmetric matrix.

`cols`

Column index of symmetric matrix.

`elems`

Nonzero elements of symmetric matrix.

COPT_GetQuadObj

Synopsis

```
int COPT_GetQuadObj(copt_prob* prob, int* p_nQElem, int* qRow, int*
qCol, double* qElem)
```

Description

Get the quadratic terms of the quadratic objective function.

Arguments

`prob`

The COPT problem.

`p_nQElem`

Pointer to number of non-zero quadratic terms . Can be `NULL`.

`qRow`

Row index of non-zero quadratic terms of the quadratic objective function.

`qCol`

Column index of non-zero quadratic terms of the quadratic objective function.

`qElem`

Values of non-zero quadratic terms of the quadratic objective function.

COPT_GetPSDObj

Synopsis

```
int COPT_GetPSDObj(copt_prob *prob, int iCol, int *p_idx)
```

Description

Get the specified PSD term of objective function.

Arguments

`prob`

The COPT problem.

`iCol`

PSD variable index.

`p_idx`

Pointer to symmetric matrix index.

COPT_GetSOSs

Synopsis

```
int COPT_GetSOSs(
    copt_prob *prob,
    int nSos,
    const int *list,
    int *sosMatBeg,
    int *sosMatCnt,
    int *sosMatIdx,
    double *sosMatWt,
    int nElemSize,
    int *pReqSize)
```

Description

Get the weight matrix of SOS constraints.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `sosMatBeg`, `sosMatCnt`, `sosMatIdx` and `sosMatWt`, we get number of non-zeros elements by `pReqSize` specified by `nSos` and `list`. Secondly, allocate sufficient memory for CRS-format matrix and call this function again to extract weight matrix. If the memory of weight matrix passed to function is not sufficient, then return the first `nElemSize` non-zero elements, and the minimal required length of non-zero elements by `pReqSize`. If `list` is `NULL`, then the first `nSos` columns will be returned.

Arguments

`prob`

The COPT problem.

`nSos`

Number of SOS constraints.

`list`

Index of SOS constraints. Can be `NULL`.

`sosMatBeg, sosMatCnt, sosMatIdx` and `sosMatWt`

Defines the weight matrix of SOS constraints in compressed row storage (CRS) format. Please see **other information** of `COPT_LoadProb` for an example of the CRS format.

`nElemSize`

Length of array for non-zero weights.

`pReqSize`

Pointer to minimal length of array for non-zero weights. Can be `NULL`.

COPT_GetCones

Synopsis

```
int COPT_GetCones(
    copt_prob *prob,
    int nCone,
    const int *list,
    int *coneBeg,
    int *coneCnt,
    int *coneIdx,
    int nElemSize,
    int *pReqSize)
```

Description

Get the matrix of Second-Order-Cone (SOC) constraints.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `coneBeg`, `coneCnt` and `coneIdx`, we get number of non-zeros elements by `pReqSize` specified by `nCone` and `list`. Secondly, allocate sufficient memory for CRS-format matrix and call this function again to extract weight matrix. If the memory of weight matrix passed to function is not sufficient, then return the first `nElemSize` non-zero elements, and the minimal required length of non-zero elements by `pReqSize`. If `list` is `NULL`, then the first `nCone` columns will be returned.

Arguments

`prob`

The COPT problem.

`nCone`

Number of SOC constraints.

`list`

Index of SOC constraints. Can be `NULL`.

`coneBeg, coneCnt, coneIdx`

Defines the matrix of SOC constraints in compressed row storage (CRS) format. Please see **other information** of `COPT_LoadProb` for an example of the CRS format.

`nElemSize`

Length of array for non-zero weights.

`pReqSize`

Pointer to minimal length of array for non-zero weights. Can be `NULL`.

COPT_GetQConstr

Synopsis

```
int COPT_GetQConstr(
    copt_prob *prob,
    int qConstrIdx,
    int *qMatRow,
    int *qMatCol,
    double *qMatElem,
    int nQElemSize,
    int *pQReqSize,
    int *rowMatIdx,
    double *rowMatElem,
    char *cRowSense,
    double *dRowBound,
    int nElemSize,
    int *pReqSize)
```

Description

Get quadratic constraint.

In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `qMatRow`, `qMatCol`, `qMatElem`, `rowMatIdx` and `rowMatElem`, we get number of non-zero quadratic terms by `pQReqSize` and number of non-zero linear terms by `pReqSize` specified by `qConstrIdx`. Secondly, allocate sufficient memory for the quadratic terms and the linear terms, and call this function again to extract the quadratic constraint. If the memory of the array of the quadratic terms passed to function is not sufficient, then return the first `nQElemSize` quadratic terms, and the minimal required length of quadratic terms by `pQReqSize`. If the memory of the array of the linear terms passed to function is not sufficient, then return the first `nElemSize` linear terms, and the minimal required length of linear terms by `pReqSize`.

Arguments

`prob`

The COPT problem.

`qConstrIdx`

Index of the quadratic constraint.

`qMatRow`

Row index of non-zero quadratic terms of the quadratic constraint (row).

`qMatCol`

Column index of non-zero quadratic terms of the quadratic constraint (row).

`qMatElem`

Values of non-zero quadratic terms of the quadratic constraint (row).

`nQElemSize`

Length of array for non-zero quadratic terms of the quadratic constraint (row).

`pQReqSize`

Pointer to minimal length of array for non-zero quadratic terms of the quadratic constraint (row). Can be `NULL`.

`rowMatIdx`

Column index of non-zero linear terms of the quadratic constraint (row).

`rowMatElem`

Values of non-zero linear terms of the quadratic constraint (row).

`cRowSense`

The sense of the quadratic constraint (row).

`dRowBound`

Right hand side of the quadratic constraint (row).

`nElemSize`

Length of array for non-zero linear terms of the quadratic constraint (row).

`pReqSize`

Pointer to minimal length of array for non-zero linear terms of the quadratic constraint (row). Can be `NULL`.

COPT_GetPSDConstr

Synopsis

```
int COPT_GetPSDConstr(
    copt_prob *prob,
    int psdConstrIdx,
    int *psdColIdx,
    int *symMatIdx,
    int nColSize,
    int *pColReqSize,
    int *rowMatIdx,
    double *rowMatElem,
    double *dRowLower,
    double *dRowUpper,
    int nElemSize,
```

```
int *pReqSize)
```

Description

Get PSD constraint.

In general, users need to call this function twice to accomplish the task. Fisrtly, by passing `NULL` to arguments `psdColIdx` and `symMatIdx`, we get number of PSD terms by `pColReqSize` specified by `psdConstrIdx`, by passing `NULL` to arguments `rowMatIdx` and `rowMatElem`, we get number of linear terms by `pReqSize` specified by `qConstrIdx`. Secondly, allocate sufficient memory for the PSD terms and the linear terms, and call this function again to extract the PSD constraint. If the memory of the array of the PSD terms passed to function is not sufficient, then return the first `nColSize` PSD terms, and the minimal required length of PSD terms by `pColReqSize`. If the memory of the array of the linear terms passed to function is not sufficient, then return the first `nElemSize` linear terms, and the minimal required length of linear terms by `pReqSize`.

Arguments

`prob`

The COPT problem.

`psdConstrIdx`

PSD constraint index.

`psdColIdx`

PSD variable index.

`symMatIdx`

Symmetric matrix index.

`nColSize`

Length of array for PSD terms of the PSD constraint.

`pColReqSize`

Pointer to minimal length of array for PSD terms of the PSD constraint. Can be `NULL`.

`rowMatIdx`

Column index of non-zero linear terms of the PSD constraint.

`rowMatElem`

Values of non-zero linear terms of the PSD constraint.

`dRowLower`

Pointer to lower bound of the PSD constraint.

`dRowUpper`

Pointer to upper bound of the PSD constraint.

`nElemSize`

Length of array for non-zero linear terms of the PSD constraint.

`pReqSize`

Pointer to minimal length of array for non-zero linear terms of the PSD constraint (row). Can be `NULL`.

COPT_GetIndicator

Synopsis

```
int COPT_GetIndicator(
    copt_prob *prob,
    int rowIdx,
    int *binColIdx,
    int *binColVal,
    int *nRowMatCnt,
    int *rowMatIdx,
    double *rowMatElem,
    char *cRowSense,
    double *dRowBound,
    int nElemSize,
    int *pReqSize)
```

Description

Get the data of an indicator constraint. In general, users need to call this function twice to accomplish the task. Firstly, by passing `NULL` to arguments `nRowMatCnt`, `rowMatIdx` and `rowMatElem`, we get number of non-zeros elements by `pReqSize` specified by `rowIdx`. Secondly, allocate sufficient memory for sparse row vector and call this function again to extract data. If the memory of sparse row vector passed to function is not sufficient, then return the first `nElemSize` non-zero elements, and the minimal required length of non-zero elements by `pReqSize`.

Arguments

`prob`

The COPT problem.

`rowIdx`

Index of the indicator constraint.

`binColIdx`

Index of the indicator variable (column).

`binColVal`

Value of the indicator variable (column).

`nRowMatCnt`

Number of non-zeros elements in the linear constraint (row).

`rowMatIdx`

Column index of non-zeros elements in the linear constraint (row).

`rowMatElem`

Values of non-zero elements in the linear constraint (row).

`cRowSense`

The sense of the linear constraint (row).

`dRowBound`

Right hand side of the linear constraint (row).

`nElemSize`

Length of array for non-zero coefficients.

`pReqSize`

Pointer to minimal length of array for non-zero coefficients. Can be `NULL`.

COPT_GetColIdx

Synopsis

```
int COPT_GetColIdx(copt_prob *prob, const char *colName, int
*p_iCol)
```

Description

Get index of column by name.

Arguments

`prob`

The COPT problem.

`colName`

Name of column.

`p_iCol`

Pointer to index of column.

COPT_GetPSDColIdx

Synopsis

```
int COPT_GetPSDColIdx(copt_prob *prob, const char *psdColName, int
*p_iPSDCol)
```

Description

Get index of PSD variable by name.

Arguments

`prob`

The COPT problem.

`psdColName`

Name of PSD variable.

`p_iPSDCol`

Pointer to index of PSD variable.

COPT_GetRowIdx

Synopsis

```
int COPT_GetRowIdx(copt_prob *prob, const char *rowName, int
*p_iRow)
```

Description

Get index of row by name.

Arguments

`prob`

The COPT problem.

`rowName`

Name of row.

`p_iRow`

Pointer to index of row.

COPT_GetQConstrIdx

Synopsis

```
int COPT_GetQConstrIdx(copt_prob *prob, const char *qConstrName, int
*p_iQConstr)
```

Description

Get index of quadratic constraint by name.

Arguments

`prob`

The COPT problem.

`qConstrName`

Name of quadratic constraint.

`p_iQConstr`

Pointer to index of quadratic constraint.

COPT_GetPSDConstrIdx

Synopsis

```
int COPT_GetPSDConstrIdx(copt_prob *prob, const char *psdConstrName,
int *p_iPSDConstr)
```

Description

Get index of PSD constraint by name.

Arguments

`prob`

The COPT problem.

`psdConstrName`

Name of PSD constraint.

`p_iPSDConstr`

Pointer to index of PSD constraint.

COPT_GetColInfo

Synopsis

```
int COPT_GetColInfo(copt_prob *prob, const char *infoName, int num,
const int *list, double *info)
```

Description

Get information of column. If `list` is `NULL`, then information of the first `num` columns will be returned.

Arguments

`prob`

The COPT problem.

`infoName`

Name of information. Please refer to *Information* for supported information.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`info`

Array of information.

COPT_GetPSDColInfo

Synopsis

```
int COPT_GetPSDColInfo(copt_prob *prob, const char *infoName, int
iCol, double *info)
```

Description

Get information of PSD variable.

Arguments

`prob`

The COPT problem.

`infoName`

Name of information. Please refer to *Information* for supported information.

`iCol`

Index of PSD variable.

`info`

Array of information.

COPT_GetRowInfo

Synopsis

```
int COPT_GetRowInfo(copt_prob *prob, const char *infoName, int num,
const int *list, double *info)
```

Description

Get information of row. If `list` is `NULL`, then information of the first `num` rows will be returned.

Arguments

`prob`

The COPT problem.

`infoName`

Name of information. Please refer to *Information* for supported information.

`num`

Number of rows.

`list`

Index of rows. Can be `NULL`.

`info`

Array of information.

COPT_GetQConstrInfo

Synopsis

```
int COPT_GetQConstrInfo(copt_prob *prob, const char *infoName, int
num, const int *list, double *info)
```

Description

Get information of quadratic constraints. If `list` is `NULL`, then information of the first `num` quadratic constraints will be returned.

Arguments

`prob`

The COPT problem.

`infoName`

Name of information. Please refer to *Information* for supported information.

`num`

Number of quadratic constraints.

`list`

Index of quadratic constraints. Can be `NULL`.

`info`

Array of information.

COPT_GetPSDConstrInfo

Synopsis

```
int COPT_GetPSDConstrInfo(copt_prob *prob, const char *infoName, int
num, const int* list, double *info)
```

Description

Get information of PSD constraints. If `list` is `NULL`, then information of the first `num` PSD constraints will be returned.

Arguments

`prob`

The COPT problem.

`infoName`

Name of information. Please refer to *Information* for supported information.

`num`

Number of PSD constraints.

`list`

Index of PSD constraints. Can be `NULL`.

`info`

Array of information.

COPT_GetColType

Synopsis

```
int COPT_GetColType(copt_prob *prob, int num, const int *list, char
*type)
```

Description

Get types of columns. If `list` is `NULL`, then types of the first `num` columns will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`type`

Types of columns.

COPT_GetColBasis

Synopsis

```
int COPT_GetColBasis(copt_prob *prob, int num, const int *list, int
*colBasis)
```

Description

Get basis status of columns. If `list` is `NULL`, then basis status of the first `num` columns will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`colBasis`

Basis status of columns.

COPT_GetRowBasis

Synopsis

```
int COPT_GetRowBasis(copt_prob *prob, int num, const int *list, int
*rowBasis)
```

Description

Get basis status of rows. If `list` is `NULL`, then basis status of the first `num` rows will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of rows.

`list`

Index of rows. Can be `NULL`.

`rowBasis`

Basis status of rows.

COPT_GetQConstrSense

Synopsis

```
int COPT_GetQConstrSense(copt_prob *prob, int num, const int *list,
char *sense)
```

Description

Get senses of quadratic constraints. If `list` is `NULL`, then types of the first `num` quadratic constraints will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of quadratic constraints.

`list`

Index of quadratic constraints. Can be `NULL`.

`sense`

Array of senses.

COPT_GetQConstrRhs

Synopsis

```
int COPT_GetQConstrRhs(copt_prob *prob, int num, const int *list,
double *rhs)
```

Description

Get RHS of quadratic constraints. If `list` is `NULL`, then types of the first `num` quadratic constraints will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of quadratic constraints.

`list`

Index of quadratic constraints. Can be `NULL`.

`rhs`

Array of RHS.

COPT_GetColName

Synopsis

```
int COPT_GetColName(copt_prob *prob, int iCol, char *buff, int
buffSize, int *pReqSize)
```

Description

Get name of column by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

Arguments

`prob`

The COPT problem.

`iCol`

Index of column.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

COPT_GetPSDColName

Synopsis

```
int COPT_GetPSDColName(copt_prob *prob, int iPSDCol, char *buff, int
buffSize, int *pReqSize)
```

Description

Get name of PSD variable by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

Arguments

`prob`

The COPT problem.

`iPSDCol`

Index of PSD variable.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

COPT_GetRowName

Synopsis

```
int COPT_GetRowName(copt_prob *prob, int iRow, char *buff, int
buffSize, int *pReqSize)
```

Description

Get name of row by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

Arguments

`prob`

The COPT problem.

`iRow`

Index of row.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

COPT_GetQConstrName

Synopsis

```
int COPT_GetQConstrName(copt_prob *prob, int iQConstr, char *buff,
int buffSize, int *pReqSize)
```

Description

Get name of quadratic constraint by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

Arguments

`prob`

The COPT problem.

`iQConstr`

Index of quadratic constraint.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

COPT_GetPSDConstrName

Synopsis

```
int COPT_GetPSDConstrName(copt_prob *prob, int iPSDConstr, char
*buff, int buffSize, int *pReqSize)
```

Description

Get name of PSD constraint by index. If memory of `buff` is not sufficient, then return the first `buffSize` length of sub-string, and the length of name requested by `pReqSize`. If `buff` is `NULL`, then we can get the length of name requested by `pReqSize`.

Arguments

`prob`

The COPT problem.

`iPSDConstr`

Index of PSD constraint.

`buff`

Buffer for storing name.

`buffSize`

Length of the buffer.

`pReqSize`

Length of the requested name. Can be `NULL`.

17.6.6 Accessing and setting parameters

COPT_SetIntParam

Synopsis

```
int COPT_SetIntParam(copt_prob *prob, const char *paramName, int
intParam)
```

Description

Sets an integer parameter.

Arguments

`prob`

The COPT problem.

`paramName`

The name of the integer parameter.

`intParam`

The value of the integer parameter.

COPT_GetIntParam, COPT_GetIntParamDef/Min/Max

Synopsis

```
int COPT_GetIntParam(copt_prob *prob, const char *paramName, int
*p_intParam)
```

```
int COPT_GetIntParamDef(copt_prob *prob, const char *paramName, int
*p_intParam)
```

```
int COPT_GetIntParamMin(copt_prob *prob, const char *paramName, int
*p_intParam)
```

```
int COPT_GetIntParamMax(copt_prob *prob, const char *paramName, int
*p_intParam)
```

Description

Gets the

current

default

minimal

maximal

value of an integer parameter.

Arguments

`prob`

The COPT problem.

`paramName`

The name of the integer parameter.

`p_intParam`

Pointer to the value of the integer parameter.

COPT_SetDblParam

Synopsis

```
int COPT_SetDblParam(copt_prob *prob, const char *paramName, double
dblParam)
```

Description

Sets a double parameter.

Arguments

`prob`

The COPT problem.

`paramName`

The name of the double parameter.

`dblParam`

The value of the double parameter.

COPT_GetDblParam, COPT_GetDblParamDef/Min/Max

Synopsis

```
int COPT_GetDblParam(copt_prob *prob, const char *paramName, double
*p_dblParam)
```

```
int COPT_GetDblParamDef(copt_prob *prob, const char *paramName,
double *p_dblParam)
```

```
int COPT_GetDblParamMin(copt_prob *prob, const char *paramName,
double *p_dblParam)
```

```
int COPT_GetDblParamMax(copt_prob *prob, const char *paramName,
double *p_dblParam)
```

Description

Gets the

current

default

minimal

maximal

value of a double parameter.

Arguments

`prob`

The COPT problem.

`paramName`

The name of the double parameter.

`p_dblParam`

Pointer to the value of the double parameter.

COPT_ResetParam

Synopsis

```
int COPT_ResetParam(copt_prob *prob)
```

Description

Reset parameters to default settings.

Arguments

`prob`

The COPT problem.

COPT_WriteParam

Synopsis

```
int COPT_WriteParam(copt_prob *prob, const char *parfilename)
```

Description

Writes user defined parameters to a file. This API function will write out all the parameters that are different from their default values.

Arguments

`prob`

The COPT problem.

`parfilename`

The path to the parameter file.

COPT_WriteParamStr

Synopsis

```
int COPT_WriteParamStr(copt_prob *prob, char *str, int nStrSize, int
*pReqSize)
```

Description

Writes the modified parameters to a string buffer.

Arguments

`prob`

The COPT problem.

`str`

String buffer of modified parameters.

`nStrSize`

The size of string buffer.

`pReqSize`

Minimum space requirement of string buffer for modified parameters.

COPT_ReadParam

Synopsis

```
int COPT_ReadParam(copt_prob *prob, const char *parfilename)
```

Description

Reads and applies parameters settings as defined in the parameter file.

Arguments

`prob`

The COPT problem.

`parfilename`

The path to the parameter file.

COPT_ReadParamStr

Synopsis

```
int COPT_ReadParamStr(copt_prob *prob, const char *strParam)
```

Description

Read parameter settings from string buffer, and set parameters in COPT.

Arguments

`prob`

The COPT problem.

`strParam`

String buffer of parameter settings.

17.6.7 Accessing attributes

COPT_GetIntAttr

Synopsis

```
int COPT_GetIntAttr(copt_prob *prob, const char *attrName, int
*p_intAttr)
```

Description

Gets the value of an integer attribute.

Arguments

`prob`

The COPT problem.

`attrName`

The name of the integer attribute.

`p_intAttr`

Pointer to the value of the integer attribute.

COPT_GetDblAttr

Synopsis

```
int COPT_GetDblAttr(copt_prob *prob, const char *attrName, int
*p_dblAttr)
```

Description

Gets the value of an double attribute.

Arguments

`prob`

The COPT problem.

`attrName`

The name of the double attribute.

`p_dblAttr`

Pointer to the value of the double attribute.

17.6.8 Logging utilities

COPT_SetLogFile

Synopsis

```
int COPT_SetLogFile(copt_prob *prob, char *logfilename)
```

Description

Set log file for the problem.

Arguments

`prob`

The COPT problem.

`logfilename`

The path to the log file.

COPT_SetLogCallback

Synopsis

```
int COPT_SetLogCallback(copt_prob *prob, void (*logcb)(char *msg,
void *userdata), void *userdata)
```

Description

Set message callback for the problem.

Arguments

`prob`

The COPT problem.

`logcb`

Callback function for message.

`userdata`

User defined data. The data will be passed to the solver without modification.

17.6.9 MIP start utilities

COPT_AddMipStart

Synopsis

```
int COPT_AddMipStart(copt_prob *prob, int num, const int *list,
double *colVal)
```

Description

Add MIP start information for the problem. If `list` is `NULL`, then information of the first `num` columns will be added.

One MIP start information will be added for each call to this function.

Arguments

`prob`

The COPT problem.

`num`

Number of variables (columns).

`list`

Index of variables (columns). Can be `NULL`.

`colVal`

MIP start information.

COPT_ReadMst

Synopsis

```
int COPT_ReadMst(copt_prob *prob, const char *mstfilename)
```

Description

Read MIP start information from file, and used as initial solution for the problem.

Arguments

`prob`

The COPT problem.

`mstfilename`

The path to the MIP start file.

COPT_WriteMst

Synopsis

```
int COPT_WriteMst(copt_prob *prob, const char *mstfilename)
```

Description

Write solution or existed MIP start information in problem to file.

Arguments

`prob`

The COPT problem.

`mstfilename`

The path to the MIP start file.

17.6.10 IIS utilities

COPT_ComputeIIS

Synopsis

```
int COPT_ComputeIIS(copt_prob *prob)
```

Description

Compute IIS (Irreducible Inconsistent Subsystem) for infeasible problem.

Arguments

`prob`

The COPT problem.

COPT_GetColLowerIIS

Synopsis

```
int COPT_GetColLowerIIS(copt_prob *prob, int num, const int *list,
int *colLowerIIS)
```

Description

Get IIS status of lower bounds of columns. If `list` is `NULL`, then IIS status of the first `num` columns will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`colLowerIIS`

IIS status of lower bounds of columns.

COPT_GetColUpperIIS

Synopsis

```
int COPT_GetColUpperIIS(copt_prob *prob, int num, const int *list,
int *colUpperIIS)
```

Description

Get IIS status of upper bounds of columns. If `list` is `NULL`, then IIS status of the first `num` columns will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of columns.

`list`

Index of columns. Can be `NULL`.

`colUpperIIS`

IIS status of upper bounds of columns.

COPT_GetRowLowerIIS

Synopsis

```
int COPT_GetRowLowerIIS(copt_prob *prob, int num, const int *list,
int *rowLowerIIS)
```

Description

Get IIS status of lower bounds of rows. If `list` is `NULL`, then IIS status of the first `num` rows will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of rows.

`list`

Index of rows. Can be `NULL`.

`rowLowerIIS`

IIS status of lower bounds of rows.

COPT_GetRowUpperIIS

Synopsis

```
int COPT_GetRowUpperIIS(copt_prob *prob, int num, const int *list,
int *rowUpperIIS)
```

Description

Get IIS status of upper bounds of rows. If `list` is `NULL`, then IIS status of the first `num` rows will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of rows.

`list`

Index of rows. Can be `NULL`.

`rowUpperIIS`

IIS status of upper bounds of rows.

COPT_GetSOSIIS

Synopsis

```
int COPT_GetSOSIIS(copt_prob *prob, int num, const int *list, int
*sosIIS)
```

Description

Get IIS status of SOS constraints. If `list` is `NULL`, then IIS status of the first `num` SOS constraints will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of SOS constraints.

`list`

Index of SOS constraints. Can be `NULL`.

`sosIIS`

IIS status of SOS constraints.

COPT_GetIndicatorIIS

Synopsis

```
int COPT_GetIndicatorIIS(copt_prob *prob, int num, const int *list,
int *indicatorIIS)
```

Description

Get IIS status of indicator constraints. If `list` is `NULL`, then IIS status of the first `num` indicator constraints will be returned.

Arguments

`prob`

The COPT problem.

`num`

Number of indicator constraints.

`list`

Index of indicator constraints. Can be `NULL`.

`indicatorIIS`

IIS status of indicator constraints.

17.6.11 Feasibility relaxation utilities

COPT_FeasRelax

Synopsis

```
int COPT_FeasRelax(copt_prob *prob, double *colLowPen, double
*colUppPen, double *rowBndPen, double *rowUppPen)
```

Description

Compute feasibility relaxation to infeasible problem.

Arguments

`prob`

The COPT problem.

`colLowPen`

Penalty for lower bounds of columns. If `NULL`, then no relaxation for lower bounds of columns are allowed; If penalty in `colLowPen` is `COPT_INFINITY`, then no relaxation is allowed for corresponding bound.

`colUppPen`

Penalty for upper bounds of columns. If `NULL`, then no relaxation for upper bounds of columns are allowed; If penalty in `colUppen` is `COPT_INFINITY`, then no relaxation is allowed for corresponding bound.

`rowBndPen`

Penalty for bounds of rows. If `NULL`, then no relaxation for bounds of rows are allowed; If penalty in `rowBndPen` is `COPT_INFINITY`, then no relaxation is allowed for corresponding bound.

`rowUppPen`

Penalty for upper bounds of rows. If problem has two-sided rows, and `rowUppPen` is not `NULL`, then `rowUppPen` is penalty for upper bounds of rows; If penalty in `rowUppPen` is `COPT_INFINITY`, then no relaxation is allowed for corresponding bound.

Note: Normally, just set `rowUppPen` to `NULL`.

COPT_WriteRelax

Synopsis

```
int COPT_WriteRelax(copt_prob *prob, const char *relaxfilename)
```

Description

Write feasrelax problem to file.

Arguments

`prob`

The COPT problem.

`relaxfilename`

Name of feasrelax problem file.

17.6.12 Parameter tuning utilities

COPT_Tune

Synopsis

```
int COPT_Tune(copt_prob *prob)
```

Description

Parameter tuning of the model.

Arguments

`prob`

COPT model.

COPT_LoadTuneParam

Synopsis

```
int COPT_LoadTuneParam(copt_prob *prob, int idx)
```

Description

Load the parameter tuning results of the specified number into the model.

Arguments

`prob`

COPT model.

`idx`

The number of the parameter tuning result.

COPT_ReadTune

Synopsis

```
int COPT_ReadTune(copt_prob *prob, const char *tunefilename)
```

Description

Read the parameter combination to be tuned from the tuning file to the model.

Arguments

`prob`

COPT model.

`tunefilename`

Tuning file names.

COPT_WriteTuneParam

Synopsis

```
int COPT_WriteTuneParam(copt_prob *prob, int idx, const char
*parfilename)
```

Description

Output the parameter tuning result of the specified number to the parameter file.

Arguments

`prob`

COPT model.

`idx`

The number of the parameter tuning result.

`parfilename`

parameter file name.

17.6.13 Callback utilities

Certain callback utilities methods can only be called in certain contexts, which are listed below:

Table 17.1: Callback utilities

Context	Methods
COPT_CBCONTEXT_MIPSOL	*COPT_AddCallbackLazyConstr* *COPT_AddCallbackLazyConstrs* *COPT_AddCallbackSolution* *COPT_GetCallbackInfo*
COPT_CBCONTEXT_MIPRELAX	*COPT_AddCallbackUserCut* *COPT_AddCallbackUserCuts* *COPT_AddCallbackLazyConstr* *COPT_AddCallbackLazyConstrs* *COPT_AddCallbackSolution* *COPT_GetCallbackInfo*

COPT_AddLazyConstr

Synopsis

```
int COPT_AddLazyConstr(
    copt_prob *prob,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense,
    double dRowBound,
    double dRowUpper,
    const char *rowName)
```

Description

Add a lazy constraint to the MIP model.

Arguments

`prob`

The COPT problem.

`nRowMatCnt`

Number of non-zero elements in the lazy constraint.

`rowMatIdx`

Column index of non-zero elements in the lazy constraint.

`rowMatElem`

Values of non-zero elements in the lazy constraint.

`cRowSense`

The sense of the new lazy constraint.

Please refer to *Constraint senses* for all the supported types.

If `cRowSense` is 0, then `dRowBound` and `dRowUpper` will be treated as lower and upper bounds for the constraint. This is the recommended method for defining constraints.

If `cRowSense` is provided, then `dRowBound` and `dRowUpper` will be treated as RHS and **range** for the constraint. In this case, `dRowUpper` is only required when `cRowSense = COPT_RANGE`, where

lower bound is `dRowBound - dRowUpper`

upper bound is `dRowBound`

`dRowBound`

Lower bound or RHS of the lazy constraint.

`dRowUpper`

Upper bound or **range** of the lazy constraint.

`rowName`

The name of the lazy constraint. Can be `NULL`.

COPT_AddLazyConstrs

Synopsis

```
int COPT_AddLazyConstrs(
    copt_prob *prob,
    int nAddRow,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const char *rowSense,
    const double *rowBound,
    const double *rowUpper,
```

```
    char const *const *rowNames)
```

Description

Add a set of lazy constraints to the MIP model.

Arguments

`prob`

The COPT problem.

`nAddRow`

Number of new lazy constraints.

`rowMatBeg, rowMatCnt, rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`rowSense`

Senses of new lazy constraints.

Please refer to *Constraint senses* for all the supported types.

If `rowSense` is `NULL`, then `rowBound` and `rowUpper` will be treated as lower and upper bounds for constraints. This is the recommended method for defining constraints.

If `rowSense` is provided, then `rowBound` and `rowUpper` will be treated as RHS and **range** for constraints. In this case, `rowUpper` is only required when there are `COPT_RANGE` constraints, where the

lower bound is `rowBound[i] - fabs(rowUpper[i])`

upper bound is `rowBound[i]`

`rowBound`

Lower bounds or RHS of new lazy constraints.

`rowUpper`

Upper bounds or **range** of new lazy constraints.

`rowNames`

Names of new lazy constraints. Can be `NULL`.

COPT_AddUserCut

Synopsis

```
int COPT_AddUserCut(
    copt_prob* prob,
    int nRowMatCnt,
    const int* rowMatIdx,
    const double* rowMmatElem,
    char cRowSense,
    double dRowBound,
    double dRowUpper,
    const char* rowName)
```

Description

Add a user cut to the MIP model.

Arguments

`prob`

The COPT problem.

`nRowMatCnt`

Number of non-zero elements in the user cut.

`rowMatIdx`

Column index of non-zero elements in the user cut.

`rowMatElem`

Values of non-zero elements in the user cut.

`cRowSense`

The sense of the user cut.

Please refer to *Constraint senses* for all the supported types.

If `cRowSense` is 0, then `dRowBound` and `dRowUpper` will be treated as lower and upper bounds for the constraint. This is the recommended method for defining constraints.

If `cRowSense` is provided, then `dRowBound` and `dRowUpper` will be treated as RHS and **range** for the constraint. In this case, `dRowUpper` is only required when `cRowSense = COPT_RANGE`, where

lower bound is `dRowBound - dRowUpper`

upper bound is `dRowBound`

`dRowBound`

Lower bound or RHS of the user cut.

`dRowUpper`

Upper bound or **range** of the user cut.

`rowName`

The name of the user cut. Can be `NULL`.

COPT_AddUserCuts

Synopsis

```
int COPT_CALL COPT_AddUserCuts(
    copt_prob *prob,
    int nAddRow,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const char *rowSense,
    const double *rowBound,
```

```
    const double *rowUpper,
    char const *const *rowNames)
```

Description

Add a set of user cuts to the MIP model.

Arguments

`prob`

The COPT problem.

`nAddRow`

Number of new user cuts.

`rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`rowSense`

Senses of new user cuts.

Please refer to *Constraint senses* for all the supported types.

If `rowSense` is `NULL`, then `rowBound` and `rowUpper` will be treated as lower and upper bounds for constraints. This is the recommended method for defining constraints.

If `rowSense` is provided, then `rowBound` and `rowUpper` will be treated as RHS and **range** for constraints. In this case, `rowUpper` is only required when there are `COPT_RANGE` constraints, where the

lower bound is `rowBound[i] - fabs(rowUpper[i])`

upper bound is `rowBound[i]`

`rowBound`

Lower bounds or RHS of new user cuts.

`rowUpper`

Upper bounds or **range** of new user cuts.

`rowNames`

Names of new user cuts. Can be `NULL`.

COPT_SetCallback

Synopsis

```
int COPT_SetCallback(
    copt_prob *prob,
    int (COPT_CALL *cb)(copt_prob *prob, void *cbdata, int cbctx,
    void *userdata),
    int cbctx,
    void *userdata)
```

Description

Set the callback function of the model.

Arguments

`prob`

The COPT problem.

`cb`

Callback function.

`cbctx`

Callback context. Please refer to *Callback context* .

`userdata`

User defined data. The data will be passed to the solver without modification.

COPT_GetCallbackInfo

Synopsis

```
int COPT_GetCallbackInfo(void *cbdata, const char* cbinfo, void
*p_val)
```

Description

Retrieve the value of the specified callback information.

Arguments

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_GetCallbackInfo().

`cbinfo`

The name of the callback information. Please refer to *Callback information* for possible values.

`p_val`

Pointer to the value of the callback information.

COPT_AddCallbackSolution

Synopsis

```
int COPT_AddCallbackSolution(void *cbdata, const double *sol,
double* p_objval)
```

Description

Set heuristic solution values for the specified variables.

Arguments

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_AddCallbackSolution().

`sol`

The solution vector.

`p_objval`

Pointer to the objective value for solution.

COPT_AddCallbackUserCut

Synopsis

```
int COPT_AddCallbackUserCut(
    void *cbdata,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense,
    double dRowRhs)
```

Description

Add a user cut to the MIP model from within the callback function.

Arguments

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_AddCallbackUserCut().

`nRowMatCnt`

Number of non-zero elements in the user cut.

`rowMatIdx`

Column index of non-zero elements in the user cut.

`rowMatElem`

Values of non-zero elements in the user cut.

`cRowSense`

The sense of the new user cut. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

The user cut added from within callback can only have a single bound.

`dRowRhs`

RHS of the user cut.

COPT_AddCallbackUserCuts

Synopsis

```
int COPT_AddCallbackUserCuts(
    void *cbdata,
    int nAddRow,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
```

```
    const double *rowMatElem,
    const char *rowSense,
    const double *rowRhs)
```

Description

Add a set of user cuts to the MIP model from within the callback function.

Arguments

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_AddCallbackUserCuts().

`nAddRow`

Number of new user cuts.

`rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`rowSense`

Senses of new user cuts. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

The user cuts added from within callback can only have single bounds.

`rowRhs`

RHS of new user cuts.

COPT_AddCallbackLazyConstr

Synopsis

```
int COPT_AddCallbackLazyConstr(
    void *cbdata,
    int nRowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    char cRowSense,
    double dRowRhs)
```

Description

Add a lazy constraint to the MIP model from within the callback function.

Arguments

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_AddCallbackLazyConstr().

`nRowMatCnt`

Number of non-zero elements in the lazy constraint.

When `nRowMatCnt<=0`, the MIP candidate solution will be directly rejected without adding a lazy constraint. And `rowMatIdx`, `rowMatElem`, `cRowSense` and `dRowRhs` will be ignored.

`rowMatIdx`

Column index of non-zero elements in the lazy constraint.

`rowMatElem`

Values of non-zero elements in the lazy constraint.

`cRowSense`

The sense of new lazy constraint. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

The lazy constraint added from within callback can only have a single bound.

`dRowRhs`

RHS of the lazy constraint.

COPT_AddCallbackLazyConstrs

Synopsis

```
int COPT_AddCallbackLazyConstrs(
    void *cbdata,
    int nAddRow,
    const int *rowMatBeg,
    const int *rowMatCnt,
    const int *rowMatIdx,
    const double *rowMatElem,
    const char *rowSense,
    const double *rowRhs)
```

Description

Add a set of lazy constraints to the MIP model from within the callback function.

Arguments

`cbdata`

The cbdata argument that was passed into the user callback by COPT. This argument must be passed unmodified from the user callback to COPT_AddCallbackLazyConstrs().

`nAddRow`

Number of new lazy constraints.

When `nAddRow<=0`, the MIP candidate solution will be directly rejected without adding lazy constraints.

And other parameters(apart from `cbdata`) will be ignored.

`rowMatBeg`, `rowMatCnt`, `rowMatIdx` and `rowMatElem`

Defines the coefficient matrix in compressed row storage (CRS) format. The CRS format is similar to the CCS format described in the **other information** of `COPT_LoadProb`.

`rowSense`

Senses of new lazy constraints. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

The lazy constraints added from within callback can only have single bounds.

`rowRhs`

RHS of new lazy constraints.

Note: If the user gives a solution to COPT in a callback, the user should ensure that this fulfills the lazy constraint callback since that won't be called for user-provided solutions.

17.6.14 Other API functions

COPT_GetBanner

Synopsis

```
int COPT_GetBanner(char *buff, int buffSize)
```

Description

Obtains a C-style string as banner, which describes the COPT version information.

Arguments

`buff`

A buffer for holding the resulting string.

`buffSize`

The size of the above buffer.

COPT_GetRetcodeMsg

Synopsis

```
int COPT_GetRetcodeMsg(int code, char *buff, int buffSize)
```

Description

Obtains a C-style string which explains a return code value in plain text.

Arguments

`code`

The return code from a COPT API function.

`buff`

A buffer for holding the resulting string.

`buffSize`

The size of the above buffer.

COPT_Interrupt

Synopsis

```
int COPT_Interrupt(copt_prob *prob)
```

Description

Interrupt solving process of current problem.

Arguments

`prob`

The COPT problem.

Chapter 18

C++ API Reference

The **Cardinal Optimizer** provides C++ API library. This chapter documents all COPT constants, including parameters and attributes, and API functions for C++ applications.

18.1 Constants

All C++ constants are the same as C constants. Please refer to *C API Reference: Constants* for more details.

18.2 Attributes

All C++ attributes are the same as C attributes. Please refer to *C API Reference: Attributes* for more details.

In the C++ API, user can get the attribute value by specifying the attribute name. The provided functions are as follows, please refer to *C++ API: Model Class* for details.

- `Model::GetIntAttr()` : Get value of a COPT integer attribute.
- `Model::GetDblAttr()` : Get value of a COPT double attribute.

18.3 Parameters

All C++ parameters are the same as C parameters. Please refer to *C API Reference: Parameters* for more details.

In the C++ API, user can get and set the parameter value by specifying the parameter name. The provided functions are as follows, please refer to *C++ Model class* for details.

- Get detailed information of the specified parameter (current value/max/min): `Model::GetParamInfo()`
- Get the current value of the specified integer/double parameter: `Model::GetIntParam()` / `Model::GetDblParam()`
- Set the specified integer/double parameter value: `Model::SetIntParam()` / `Model::SetDblParam()`

18.4 C++ Modeling Classes

This chapter documents COPT C++ interface. Users may refer to C++ classes described below for details of how to construct and solve C++ models.

18.4.1 Envr

Essentially, any C++ application using Cardinal Optimizer should start with a COPT environment. COPT models are always associated with a COPT environment. User must create an environment object before populating models. User generally only need a single environment object in program.

Envr::Envr()

Constructor of COPT Envr object.

Synopsis

```
Envr()
```

Envr::Envr()

Constructor of COPT Envr object, given a license folder.

Synopsis

```
Envr(const char *szLicDir)
```

Arguments

`szLicDir`: directory having local license or client config file.

Envr::Envr()

Constructor of COPT Envr object, given an Envr config object.

Synopsis

```
Envr(const EnvrConfig &config)
```

Arguments

`config`: Envr config object holding settings for remote connection.

Envr::Close()

close remote connection and token becomes invalid for all problems in current envr.

Synopsis

```
void Close()
```

Envr::CreateModel()

Create a COPT model object.

Synopsis

```
Model CreateModel(const char *szName)
```

Arguments

`szName`: customized model name.

Return

a COPT model object.

18.4.2 EnvrConfig

If user connects to COPT remote services, such as floating token server or compute cluster, it is necessary to add config settings with EnvrConfig object.

EnvrConfig::EnvrConfig()

Constructor of COPT environment config object.

Synopsis

```
EnvrConfig()
```

EnvrConfig::Set()

Set config settings in terms of name-value pair.

Synopsis

```
void Set(const char *szName, const char *szValue)
```

Arguments

`szName`: keyword of a config setting

`szValue`: value of a config setting

18.4.3 Model

In general, a COPT model consists of a set of variables, a (linear) objective function on these variables, a set of constraints on there varaibles, etc. COPT model class encapsulates all required methods for constructing a COPT model.

Model::AddCone()

Add a cone constraint to a model, given its dimension.

Synopsis

```
Cone AddCone(
    int dim,
    int type,
    char *pvtype,
    const char *szPrefix)
```

Arguments

`dim`: dimension of the cone constraint.

`type`: type of the cone constraint.

`pvtype`: types of variables in the cone.

`szPrefix`: name prefix of variables in the cone.

Return

object of new cone constraint.

Model::AddCone()

Add a cone constraint to model.

Synopsis

```
Cone AddCone(const ConeBuilder &builder)
```

Arguments

`builder`: builder for new cone constraint.

Return

new cone constraint object.

Model::AddCone()

Add a cone constraint to model.

Synopsis

```
Cone AddCone(const VarArray &vars, int type)
```

Arguments

`vars`: variables that participate in the cone constraint.

`type`: type of the cone constraint.

Return

object of new cone constraint.

Model::AddConstr()

Add a linear constraint to model.

Synopsis

```
Constraint AddConstr(
    const Expr &expr,
    char sense,
    double rhs,
    const char *szName)
```

Arguments

`expr`: expression for the new contraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side value for the new constraint.

`szName`: optional, name of new constraint.

Return

new constraint object.

Model::AddConstr()

Add a linear constraint to model.

Synopsis

```
Constraint AddConstr(
    const Expr &lhs,
    char sense,
    const Expr &rhs,
    const char *szName)
```

Arguments

`lhs`: left hand side expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side expression for the new constraint.

`szName`: optional, name of new constraint.

Return

new constraint object.

Model::AddConstr()

Add a linear constraint to model.

Synopsis

```
Constraint AddConstr(
    const Expr &expr,
    double lb,
    double ub,
    const char *szName)
```

Arguments

`expr`: expression for the new constraint.

`lb`: lower bound for the new constraint.

`ub`: upper bound for the new constraint

`szName`: optional, name of new constraint.

Return

new constraint object.

Model::AddConstr()

Add a linear constraint to a model.

Synopsis

```
Constraint AddConstr(const ConstrBuilder &builder, const char
*szName)
```

Arguments

`builder`: builder for the new constraint.

`szName`: optional, name of new constraint.

Return

new constraint object.

Model::AddConstrs()

Add linear constraints to model.

Synopsis

```
ConstrArray AddConstrs(
    int count,
    char *pSense,
    double *pRhs,
    const char *szPrefix)
```

Arguments

`count`: number of constraints added to model.

`pSense`: sense array for new linear constraints, other than range sense.

`pRhs`: right hand side values for new variables.

`szPrefix`: name prefix for new constraints.

Return

array of new constraint objects.

Model::AddConstrs()

Add linear constraints to a model.

Synopsis

```
ConstrArray AddConstrs(
    int count,
    double *pLower,
    double *pUpper,
    const char *szPrefix)
```

Arguments

`count`: number of constraints added to the model.

`pLower`: lower bounds of new constraints.

`pUpper`: upper bounds of new constraints.

`szPrefix`: name prefix for new constraints.

Return

array of new constraint objects.

Model::AddConstrs()

Add linear constraints to a model.

Synopsis

```
ConstrArray AddConstrs(
    int count,
    double *pLower,
    double *pUpper,
    const char *szNames,
    size_t len)
```

Arguments

`count`: number of constraints added to the model.

`pLower`: lower bounds of new constraints.

`pUpper`: upper bounds of new constraints.

`szNames`: name buffer of new constraints.

`len`: length of the name buffer.

Return

array of new constraint objects.

Model::AddConstrs()

Add linear constraints to a model.

Synopsis

```
ConstrArray AddConstrs(const ConstrBuilderArray &builders, const
char *szPrefix)
```

Arguments

`builders`: builders for new constraints.

`szPrefix`: name prefix for new constraints.

Return

array of new constraint objects.

Model::AddConstrs()

Add linear constraints to model.

Synopsis

```
ConstrArray AddConstrs(
    const ConstrBuilderArray &builders,
    const char *szNames,
    size_t len)
```

Arguments

`builders`: builders for new constraints.

`szNames`: name buffer of new constraints.

`len`: length of the name buffer.

Return

array of new constraint objects.

Model::AddDenseMat()

Add a dense symmetric matrix to a model.

Synopsis

```
SymMatrix AddDenseMat(
    int dim,
    double *pVals,
    int len)
```

Arguments

`dim`: dimension of the dense symmetric matrix.

`pVals`: array of non-zero elements, filled in column-wise up to len or max length of symmetric matrix.

`len`: length of value array.

Return

new symmetric matrix object.

Model::AddDenseMat()

Add a dense symmetric matrix to a model.

Synopsis

```
SymMatrix AddDenseMat(int dim, double val)
```

Arguments

`dim`: dimension of dense symmetric matrix.

`val`: value to fill dense symmetric matrix.

Return

new symmetric matrix object.

Model::AddDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix AddDiagMat(int dim, double val)
```

Arguments

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

Return

new diagonal matrix object.

Model::AddDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix AddDiagMat(
    int dim,
    double *pVals,
    int len)
```

Arguments

`dim`: dimension of diagonal matrix.

`pVals`: array of values of diagonal elements.

`len`: length of value array.

Return

new diagonal matrix object.

Model::AddDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix AddDiagMat(
    int dim,
    double val,
    int offset)
```

Arguments

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

`offset`: shift distance against diagonal line.

Return

new diagonal matrix object.

Model::AddDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix AddDiagMat(
    int dim,
    double *pVals,
    int len,
    int offset)
```

Arguments

`dim`: dimension of diagonal matrix.

`pVals`: array of values of diagonal elements.

`len`: length of value array.

`offset`: shift distance against diagonal line.

Return

new diagonal matrix object.

Model::AddEyeMat()

Add an identity matrix to a model.

Synopsis

```
SymMatrix AddEyeMat(int dim)
```

Arguments

`dim`: dimension of identity matrix.

Return

new identity matrix object.

Model::AddGenConstrIndicator()

Add a general constraint of type indicator to model.

Synopsis

```
GenConstr AddGenConstrIndicator(const GenConstrBuilder &builder)
```

Arguments

`builder`: builder for the general constraint.

Return

new general constraint object of type indicator.

Model::AddGenConstrIndicator()

Add a general constraint of type indicator to model.

Synopsis

```
GenConstr AddGenConstrIndicator(
    Var binVar,
    int binVal,
    const ConstrBuilder &constr)
```

Arguments

`binVar`: binary indicator variable.

`binVal`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`constr`: builder for linear constraint.

Return

new general constraint object of type indicator.

Model::AddGenConstrIndicator()

Add a general constraint of type indicator to model.

Synopsis

```
GenConstr AddGenConstrIndicator(
    Var binVar,
    int binVal,
    const Expr &expr,
    char sense,
    double rhs)
```

Arguments

`binVar`: binary indicator variable.

`binVal`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`expr`: expression for new linear contraint.

`sense`: sense for new linear constraint.

`rhs`: right hand side value for new linear constraint.

Return

new general constraint object of type indicator.

Model::AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(
    const Expr &lhs,
    char sense,
    double rhs,
    const char *szName)
```

Arguments

`lhs`: expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

`szName`: optional, name of lazy constraint.

Model::AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(
    const Expr &lhs,
    char sense,
    const Expr &rhs,
    const char *szName)
```

Arguments

`lhs`: left hand side expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy contraint.

`szName`: optional, name of lazy constraint.

Model::AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(const ConstrBuilder &builder, const char *szName)
```

Arguments

`builder`: builder for lazy contraint.

`szName`: optional, name of lazy constraint.

Model::AddLazyConstrs()

Add lazy constraints to model.

Synopsis

```
void AddLazyConstrs(const ConstrBuilderArray &builders, const char
*szPrefix)
```

Arguments

`builders`: array of builders for lazy contraints.

`szPrefix`: name prefix of new lazy constraints.

Model::AddOnesMat()

Add a dense symmetric matrix of value one to a model.

Synopsis

```
SymMatrix AddOnesMat(int dim)
```

Arguments

`dim`: dimension of dense symmetric matrix.

Return

new symmetric matrix object.

Model::AddPsdConstr()

Add a PSD constraint to model.

Synopsis

```
PsdConstraint AddPsdConstr(
    const PsdExpr &expr,
    char sense,
    double rhs,
    const char *szName)
```

Arguments

`expr`: PSD expression for new PSD contraint.

`sense`: sense for new PSD constraint.

`rhs`: double value at right side of the new PSD constraint.

`szName`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model::AddPsdConstr()

Add a PSD constraint to model.

Synopsis

```
PsdConstraint AddPsdConstr(
    const PsdExpr &expr,
    double lb,
    double ub,
    const char *szName)
```

Arguments

`expr`: expression for new PSD constraint.

`lb`: lower bound for ew PSD constraint.

`ub`: upper bound for new PSD constraint

`szName`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model::AddPsdConstr()

Add a PSD constraint to model.

Synopsis

```
PsdConstraint AddPsdConstr(
    const PsdExpr &lhs,
    char sense,
    const PsdExpr &rhs,
    const char *szName)
```

Arguments

`lhs`: PSD expression at left side of new PSD constraint.

`sense`: sense for new PSD constraint.

`rhs`: PSD expression at right side of new PSD constraint.

`szName`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model::AddPsdConstr()

Add a PSD constraint to a model.

Synopsis

```
PsdConstraint AddPsdConstr(const PsdConstrBuilder &builder, const
char *szName)
```

Arguments

`builder`: builder for new PSD constraint.

`szName`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model::AddPsdVar()

Add a new PSD variable to model.

Synopsis

```
PsdVar AddPsdVar(int dim, const char *szName)
```

Arguments

`dim`: dimension of new PSD variable.

`szName`: name of new PSD variable.

Return

PSD variable object.

Model::AddPsdVars()

Add new PSD variables to model.

Synopsis

```
PsdVarArray AddPsdVars(
    int count,
    int *pDim,
    const char *szPrefix)
```

Arguments

`count`: number of new PSD variables.

`pDim`: array of dimensions of new PSD variables.

`szPrefix`: name prefix of new PSD variables.

Return

array of PSD variable objects.

Model::AddPsdVars()

Add new PSD variables to model.

Synopsis

```
PsdVarArray AddPsdVars(
    int count,
    int *pDim,
    const char *szNames,
    size_t len)
```

Arguments

`count`: number of new PSD variables.

`pDim`: array of dimensions of new PSD variables.

`szNames`: name buffer of new PSD variables.

`len`: length of the name buffer.

Return

array of PSD variable objects.

Model::AddQConstr()

Add a quadratic constraint to model.

Synopsis

```
QConstraint AddQConstr(
    const QuadExpr &expr,
    char sense,
    double rhs,
    const char *szName)
```

Arguments

`expr`: quadratic expression for the new contraint.

`sense`: sense for new quadratic constraint.

`rhs`: double value at right side of the new quadratic constraint.

`szName`: optional, name of new quadratic constraint.

Return

new quadratic constraint object.

Model::AddQConstr()

Add a quadratic constraint to model.

Synopsis

```
QConstraint AddQConstr(
    const QuadExpr &lhs,
    char sense,
    const QuadExpr &rhs,
    const char *szName)
```

Arguments

`lhs`: quadratic expression at left side of the new quadratic constraint.

`sense`: sense for new quadratic constraint.

`rhs`: quadratic expression at right side of the new quadratic constraint.

`szName`: optional, name of new quadratic constraint.

Return

new quadratic constraint object.

Model::AddQConstr()

Add a quadratic constraint to a model.

Synopsis

```
QConstraint AddQConstr(const QConstrBuilder &builder, const char
*szName)
```

Arguments

`builder`: builder for the new quadratic constraint.

`szName`: optional, name of new quadratic constraint.

Return

new quadratic constraint object.

Model::AddSos()

Add a SOS constraint to model.

Synopsis

```
Sos AddSos(const SosBuilder &builder)
```

Arguments

`builder`: builder for new SOS constraint.

Return

new SOS constraint object.

Model::AddSos()

Add a SOS constraint to model.

Synopsis

```
Sos AddSos(
    const VarArray &vars,
    const double *pWeights,
    int type)
```

Arguments

`vars`: variables that participate in the SOS constraint.

`pWeights`: optional, weights for variables in the SOS constraint.

`type`: type of SOS constraint.

Return

new SOS constraint object.

Model::AddSparseMat()

Add a sparse symmetric matrix to a model.

Synopsis

```
SymMatrix AddSparseMat(
    int dim,
    int nElems,
    int *pRows,
    int *pCols,
    double *pVals)
```

Arguments

`dim`: dimension of the sparse symmetric matrix.

`nElems`: number of non-zero elements in the sparse symmetric matrix.

`pRows`: array of row indexes of non-zero elements.

`pCols`: array of col indexes of non-zero elements.

`pVals`: array of values of non-zero elements.

Return

new symmetric matrix object.

Model::AddSymMat()

Given a symmetric matrix expression, add results matrix to model.

Synopsis

```
SymMatrix AddSymMat(const SymMatExpr &expr)
```

Arguments

`expr`: symmetric matrix expression object.

Return

results symmetric matrix object.

Model::AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(
    const Expr &lhs,
    char sense,
    double rhs,
    const char *szName)
```

Arguments

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

`szName`: optional, name of user cut.

Model::AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(
    const Expr &lhs,
    char sense,
    const Expr &rhs,
    const char *szName)
```

Arguments

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

`szName`: optional, name of user cut.

Model::AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(const ConstrBuilder &builder, const char *szName)
```

Arguments

`builder`: builder for user cut.

`szName`: optional, name of user cut.

Model::AddUserCuts()

Add user cuts to model.

Synopsis

```
void AddUserCuts(const ConstrBuilderArray &builders, const char
*szPrefix)
```

Arguments

`builders`: array of builders for user cuts.

`szPrefix`: name prefix of new user cuts.

Model::AddVar()

Add a variable to model.

Synopsis

```
Var AddVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    const char *szName)
```

Arguments

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`szName`: optional, name for new variable.

Return

new variable object.

Model::AddVar()

Add a variable and the associated non-zero coefficients as column.

Synopsis

```
Var AddVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    const Column &col,
    const char *szName)
```

Arguments

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`col`: column object for specifying a set of constraints to which the variable belongs.

`szName`: optional, name for new variable.

Return

new variable object.

Model::AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    int count,
    char vtype,
    const char *szPrefix)
```

Arguments

`count`: the number of variables to add.

`vtype`: variable types for new variables.

`szPrefix`: prefix part for names of new variables.

Return

array of new variable objects.

Model::AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    int count,
    char vtype,
    const char *szNames,
    size_t len)
```

Arguments

`count`: the number of variables to add.

`vtype`: variable types for new variables.

`szNames`: name buffer for new variables.

`len`: length of name buffer.

Return

array of new variable objects.

Model::AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    int count,
    double lb,
    double ub,
    double obj,
    char vtype,
    const char *szPrefix)
```

Arguments

`count`: the number of variables to add.

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable types for new variables.

`szPrefix`: prefix part for names of new variables.

Return

array of new variable objects.

Model::AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    int count,
    double lb,
    double ub,
    double obj,
    char vtype,
    const char *szNames,
    size_t len)
```

Arguments

`count`: the number of variables to add.

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable types for new variables.

`szNames`: name buffer for new variables.

`len`: length of name buffer.

Return

array of new variable objects.

Model::AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    int count,
    double *plb,
    double *pub,
    double *pobj,
    char *pvtype,
    const char *szPrefix)
```

Arguments

`count`: the number of variables to add.

`plb`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`pub`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`pobj`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`pvtype`: variable types for new variables. if NULL, variable types are continuous.

`szPrefix`: prefix part for names of new variables.

Return

array of new variable objects.

Model::AddVars()

Add new decision variables to a model.

Synopsis

```
VarArray AddVars(
    int count,
    double *plb,
    double *pub,
    double *pobj,
    char *pvtype,
    const char *szNames,
    size_t len)
```

Arguments

`count`: the number of variables to add.

`plb`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`pub`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`pobj`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`pvtype`: variable types for new variables. if NULL, variable types are continuous.

`szNames`: name buffer for new variables.

`len`: length of name buffer.

Return

array of new variable objects.

Model::AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    int count,
    double *plb,
    double *pub,
    double *pobj,
    char *pvtype,
    const ColumnArray &cols,
```

```
    const char *szPrefix)
```

Arguments

`count`: the number of variables to add.

`plb`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`pub`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`pobj`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`pvtype`: variable types for new variables. if NULL, variable types are continuous.

`cols`: column objects for specifying a set of constraints to which each new variable belongs.

`szPrefix`: prefix part for names of new variables.

Return

array of new variable objects.

Model::Clear()

Clear all settings including problem itself.

Synopsis

```
void Clear()
```

Model::Clone()

Deep copy a new model object.

Synopsis

```
Model Clone()
```

Return

cloned model object.

Model::ComputeIIS()

Compute IIS for infeasible model.

Synopsis

```
void ComputeIIS()
```

Model::DelPsdObj()

delete PSD part of objective in model.

Synopsis

```
void DelPsdObj()
```

Model::DelQuadObj()

delete quadratic part of objective in model.

Synopsis

```
void DelQuadObj()
```

Model::FeasRelax()

Compute feasibility relaxation for infeasible model.

Synopsis

```
void FeasRelax(
    VarArray &vars,
    double *pColLowPen,
    double *pColUppPen,
    ConstrArray &cons,
    double *pRowBndPen,
    double *pRowUppPen)
```

Arguments

`vars`: an array of variables.

`pColLowPen`: penalties for lower bounds of variables.

`pColUppPen`: penalties for upper bounds of variables.

`cons`: an array of constraints.

`pRowBndPen`: penalties for right hand sides of constraints.

`pRowUppPen`: penalties for upper bounds of range constraints.

Model::FeasRelax()

Compute feasibility relaxation for infeasible model.

Synopsis

```
void FeasRelax(int ifRelaxVars, int ifRelaxCons)
```

Arguments

`ifRelaxVars`: whether to relax variables.

`ifRelaxCons`: whether to relax constraints.

Model::Get()

Query values of double parameter or attribute, associated with variables.

Synopsis

```
int Get(
    const char *szName,
    const VarArray &vars,
    double *pOut)
```

Arguments

`szName`: name of double parameter or double attribute.

`vars`: a list of desired variables.

`pOut`: output array of parameter or attribute values.

Return

the number of valid variables. If failed, return -1.

Model::Get()

Query values of parameter or attribute, associated with constraints.

Synopsis

```
int Get(
    const char *szName,
    const ConstrArray &constrs,
    double *pOut)
```

Arguments

`szName`: name of double parameter or double attribute.

`constrs`: a list of desired constraints.

`pOut`: output array of parameter or attribute values.

Return

the number of valid constraints. If failed, return -1.

Model::Get()

Query values of parameter or attribute, associated with quadratic constraints.

Synopsis

```
int Get(
    const char *szName,
    const QConstrArray &constrs,
    double *pOut)
```

Arguments

`szName`: name of double parameter or double attribute.

`constrs`: a list of desired quadratic constraints.

`pOut`: output array of parameter or attribute values.

Return

the number of valid quadratic constraints. If failed, return -1.

Model::Get()

Query values of parameter or attribute, associated with PSD constraints.

Synopsis

```
int Get(
    const char *szName,
    const PsdConstrArray &constrs,
    double *pOut)
```

Arguments

`szName`: name of double parameter or attribute.

`constrs`: a list of desired PSD constraints.

`pOut`: output array of parameter or attribute values.

Return

the number of valid PSD constraints. If failed, return -1.

Model::GetCoeff()

Get the coefficient of variable in a linear constraint.

Synopsis

```
double GetCoeff(const Constraint &constr, const Var &var)
```

Arguments

`constr`: The requested constraint.

`var`: The requested variable.

Return

The requested coefficient.

Model::GetCol()

Get a column object that have a list of constraints in which the variable participates.

Synopsis

```
Column GetCol(const Var &var)
```

Arguments

`var`: a variable object.

Return

a column object associated with a variable.

Model::GetColBasis()

Get status of column basis.

Synopsis

```
int GetColBasis(int *pBasis)
```

Arguments

`pBasis`: integer pointer to basis status.

Return

number of columns.

Model::GetCone()

Get a cone constraint of given index in model.

Synopsis

```
Cone GetCone(int idx)
```

Arguments

`idx`: index of the desired cone constraint.

Return

the desired cone constraint object.

Model::GetConeBuilders()

Get builders of all cone constraints in model.

Synopsis

```
ConeBuilderArray GetConeBuilders()
```

Return

array object of all cone constraint builders.

Model::GetConeBuilders()

Get builders of given cone constraints in model.

Synopsis

```
ConeBuilderArray GetConeBuilders(const ConeArray &cones)
```

Arguments

`cones`: array of cone constraints.

Return

array object of desired cone constraint builders.

Model::GetCones()

Get all cone constraints in model.

Synopsis

```
ConeArray GetCones()
```

Return

array object of cone constraints.

Model::GetConstr()

Get a constraint of given index in model.

Synopsis

```
Constraint GetConstr(int idx)
```

Arguments

`idx`: index of the desired constraint.

Return

the desired constraint object.

Model::GetConstrBuilder()

Get builder of a constraint in model, including variables and associated coefficients, sense and RHS.

Synopsis

```
ConstrBuilder GetConstrBuilder(Constraint constr)
```

Arguments

`constr`: a constraint object.

Return

constraint builder object.

Model::GetConstrBuilders()

Get builders of all constraints in model.

Synopsis

```
ConstrBuilderArray GetConstrBuilders()
```

Return

array object of constraint builders.

Model::GetConstrByName()

Get a constraint of given name in model.

Synopsis

```
Constraint GetConstrByName(const char *szName)
```

Arguments

`szName`: name of the desired constraint.

Return

the desired constraint object.

Model::GetConstrLowerIIS()

Get IIS status of lower bounds of constraints.

Synopsis

```
int GetConstrLowerIIS(const ConstrArray &constrs, int *pLowerIIS)
```

Arguments

`constrs`: Array of constraints.

`pLowerIIS`: IIS status of lower bounds of constraints.

Return

Number of constraints.

Model::GetConstrs()

Get all constraints in model.

Synopsis

```
ConstrArray GetConstrs()
```

Return

array object of constraints.

Model::GetConstrUpperIIS()

Get IIS status of upper bounds of constraints.

Synopsis

```
int GetConstrUpperIIS(const ConstrArray &constrs, int *pUpperIIS)
```

Arguments

`constrs`: Array of constraints.

`pUpperIIS`: IIS status of upper bounds of constraints.

Return

Number of constraints.

Model::GetDblAttr()

Get value of a COPT double attribute.

Synopsis

```
double GetDblAttr(const char *szAttr)
```

Arguments

`szAttr`: name of double attribute.

Return

value of double attribute.

Model::GetDblParam()

Get value of a COPT double parameter.

Synopsis

```
double GetDblParam(const char *szParam)
```

Arguments

`szParam`: name of integer parameter.

Return

value of double parameter.

Model::GetGenConstrIndicator()

Get builder of given general constraint of type indicator.

Synopsis

```
GenConstrBuilder GetGenConstrIndicator(const GenConstr &indicator)
```

Arguments

`indicator`: a general constraint of type indicator.

Return

builder object of general constraint of type indicator.

Model::GetIndicatorIIS()

Get IIS status of indicator constraints.

Synopsis

```
int GetIndicatorIIS(const GenConstrArray &genconstrs, int *pIIS)
```

Arguments

`genconstrs`: Array of indicator constraints.

`pIIS`: IIS status of indicator constraints.

Return

Number of indicator constraints.

Model::GetIntAttr()

Get value of a COPT integer attribute.

Synopsis

```
int GetIntAttr(const char *szAttr)
```

Arguments

`szAttr`: name of integer attribute.

Return

value of integer attribute.

Model::GetIntParam()

Get value of a COPT integer parameter.

Synopsis

```
int GetIntParam(const char *szParam)
```

Arguments

`szParam`: name of integer parameter.

Return

value of integer parameter.

Model::GetLpSolution()

Get LP solution.

Synopsis

```
void GetLpSolution(
    double *pValue,
    double *pSlack,
    double *pRowDual,
    double *pRedCost)
```

Arguments

`pValue`: optional, double pointer to solution values.

`pSlack`: optional, double poitner to slack values.

`pRowDual`: optional, double pointer to dual values.

`pRedCost`: optional, double pointer to reduced costs.

Model::GetObjective()

Get linear expression of objective for model.

Synopsis

```
Expr GetObjective()
```

Return

a linear expression object.

Model::GetParamAttrType()

Get type of a COPT parameter or attribute.

Synopsis

```
int GetParamAttrType(const char *szName)
```

Arguments

`szName`: name of parameter or attribute.

Return

type of parameter or attribute.

Model::GetParamInfo()

Get current, default, minimum, maximum of COPT integer parameter.

Synopsis

```
void GetParamInfo(
    const char *szParam,
    int *pnCur,
    int *pnDef,
    int *pnMin,
    int *pnMax)
```

Arguments

`szParam`: name of integer parameter.

`pnCur`: out, current value of integer parameter.

`pnDef`: out, default value of integer parameter.

`pnMin`: out, minimum value of integer parameter.

`pnMax`: out, maximum value of integer parameter.

Model::GetParamInfo()

Get current, default, minimum, maximum of COPT double parameter.

Synopsis

```
void GetParamInfo(
    const char *szParam,
    double *pdCur,
    double *pdDef,
    double *pdMin,
    double *pdMax)
```

Arguments

`szParam`: name of double parameter.

`pdCur`: out, current value of double parameter.

`pdDef`: out, default value of double parameter.

`pdMin`: out, minimum value of double parameter.

`pdMax`: out, maximum value of double parameter.

Model::GetPoolObjVal()

Get the iSol-th objective value in solution pool.

Synopsis

```
double GetPoolObjVal(int iSol)
```

Arguments

`iSol`: Index of solution.

Return

The requested objective value.

Model::GetPoolSolution()

Get the iSol-th solution in solution pool.

Synopsis

```
int GetPoolSolution(
    int iSol,
    const VarArray &vars,
    double *pColVals)
```

Arguments

`iSol`: Index of solution.

`vars`: The requested variables.

`pColVals`: Pointer to the requested solutions.

Return

The length of requested solution array.

Model::GetPsdCoeff()

Get the symmetric matrix of PSD variable in a PSD constraint.

Synopsis

```
SymMatrix GetPsdCoeff(const PsdConstraint &constr, const PsdVar
&var)
```

Arguments

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

Return

The associated coefficient matrix.

Model::GetPsdConstr()

Get a quadratic constraint of given index in model.

Synopsis

```
PsdConstraint GetPsdConstr(int idx)
```

Arguments

`idx`: index of the desired quadratic constraint.

Return

the desired quadratic constraint object.

Model::GetPsdConstrBuilder()

Get builder of a PSD constraint in model, including PSD variables, sense and associated symmtric matrix.

Synopsis

```
PsdConstrBuilder GetPsdConstrBuilder(const PsdConstraint &constr)
```

Arguments

`constr`: a PSD constraint object.

Return

PSD constraint builder object.

Model::GetPsdConstrBuilders()

Get builders of all PSD constraints in model.

Synopsis

```
PsdConstrBuilderArray GetPsdConstrBuilders()
```

Return

array object of PSD constraint builders.

Model::GetPsdConstrByName()

Get a quadratic constraint of given name in model.

Synopsis

```
PsdConstraint GetPsdConstrByName(const char *szName)
```

Arguments

`szName`: name of the desired constraint.

Return

the desired quadratic constraint object.

Model::GetPsdConstrs()

Get all PSD constraints in model.

Synopsis

```
PsdConstrArray GetPsdConstrs()
```

Return

array object of PSD constraints.

Model::GetPsdObjective()

Get PSD objective of model.

Synopsis

```
PsdExpr GetPsdObjective()
```

Return

a PSD expression object.

Model::GetPsdRow()

Get PSD variables and associated symmetric matrix that participate in a PSD constraint.

Synopsis

```
PsdExpr GetPsdRow(const PsdConstraint &constr)
```

Arguments

`constr`: a PSD constraint object.

Return

PSD expression object of the PSD constraint.

Model::GetPsdRow()

Get PSD variables, associated symmetric matrix, LB/UB that participate in a PSD constraint.

Synopsis

```
PsdExpr GetPsdRow(
    const PsdConstraint &constr,
    double *pLower,
    double *pUpper)
```

Arguments

`constr`: a PSD constraint object.

`pLower`: pointer to double value of lower bound.

`pUpper`: pointer to double value of upper bound.

Return

PSD expression object of the PSD constraint.

Model::GetPsdVar()

Get a PSD variable of given index in model.

Synopsis

```
PsdVar GetPsdVar(int idx)
```

Arguments

`idx`: index of the desired PSD variable.

Return

the desired PSD variable object.

Model::GetPsdVarByName()

Get a PSD variable of given name in model.

Synopsis

```
PsdVar GetPsdVarByName(const char *szName)
```

Arguments

`szName`: name of the desired PSD variable.

Return

the desired PSD variable object.

Model::GetPsdVars()

Get all PSD variables in model.

Synopsis

```
PsdVarArray GetPsdVars()
```

Return

array object of PSD variables.

Model::GetQConstr()

Get a quadratic constraint of given index in model.

Synopsis

```
QConstraint GetQConstr(int idx)
```

Arguments

`idx`: index of the desired quadratic constraint.

Return

the desired quadratic constraint object.

Model::GetQConstrBuilder()

Get builder of a constraint in model, including variables and associated coefficients, sense and RHS.

Synopsis

```
QConstrBuilder GetQConstrBuilder(const QConstraint &constr)
```

Arguments

`constr`: a constraint object.

Return

constraint builder object.

Model::GetQConstrBuilders()

Get builders of all constraints in model.

Synopsis

```
QConstrBuilderArray GetQConstrBuilders()
```

Return

array object of constraint builders.

Model::GetQConstrByName()

Get a quadratic constraint of given name in model.

Synopsis

```
QConstraint GetQConstrByName(const char *szName)
```

Arguments

`szName`: name of the desired constraint.

Return

the desired quadratic constraint object.

Model::GetQConstrs()

Get all quadratic constraints in model.

Synopsis

```
QConstrArray GetQConstrs()
```

Return

array object of quadratic constraints.

Model::GetQuadObjective()

Get quadratic objective of model.

Synopsis

```
QuadExpr GetQuadObjective()
```

Return

a quadratic expression object.

Model::GetQuadRow()

Get two variables and associated coefficients that participate in a quadratic constraint.

Synopsis

```
QuadExpr GetQuadRow(const QConstraint &constr)
```

Arguments

`constr`: a quadratic constraint object.

Return

quadratic expression object of the constraint.

Model::GetQuadRow()

Get two variables and associated coefficients that participate in a quadratic constraint.

Synopsis

```
QuadExpr GetQuadRow(
    const QConstraint &constr,
    char *pSense,
    double *pBound)
```

Arguments

`constr`: a quadratic constraint object.

`pSense`: sense of quadratic constraint.

`pBound`: right hand side of quadratic constraint.

Return

quadratic expression object of the constraint.

Model::GetRow()

Get variables that participate in a constraint, and the associated coefficients.

Synopsis

```
Expr GetRow(const Constraint &constr)
```

Arguments

`constr`: a constraint object.

Return

expression object of the constraint.

Model::GetRowBasis()

Get status of row basis.

Synopsis

```
int GetRowBasis(int *pBasis)
```

Arguments

`pBasis`: integer pointer to basis status.

Return

number of rows.

Model::GetSolution()

Get MIP solution.

Synopsis

```
void GetSolution(double *pValue)
```

Arguments

`pValue`: double pointer to solution values.

Model::GetSos()

Get a SOS constraint of given index in model.

Synopsis

```
Sos GetSos(int idx)
```

Arguments

`idx`: index of the desired SOS constraint.

Return

the desired SOS constraint object.

Model::GetSosBuilders()

Get builders of all SOS constraints in model.

Synopsis

```
SosBuilderArray GetSosBuilders()
```

Return

array object of SOS constraint builders.

Model::GetSosBuilders()

Get builders of given SOS constraints in model.

Synopsis

```
SosBuilderArray GetSosBuilders(const SosArray &soss)
```

Arguments

`soss`: array of SOS constraints.

Return

array object of desired SOS constraint builders.

Model::GetSOSIIS()

Get IIS status of SOS constraints.

Synopsis

```
int GetSOSIIS(const SosArray &soss, int *pIIS)
```

Arguments

`soss`: Array of SOS constraints.

`pIIS`: IIS status of SOS constraints.

Return

Number of SOS constraints.

Model::GetSoss()

Get all SOS constraints in model.

Synopsis

```
SosArray GetSoss()
```

Return

array object of SOS constraints.

Model::GetSymMat()

Get a symmetric matrix of given index in model.

Synopsis

```
SymMatrix GetSymMat(int idx)
```

Arguments

`idx`: index of the desired symmetric matrix.

Return

the desired symmetric matrix object.

Model::GetVar()

Get a variable of given index in model.

Synopsis

```
Var GetVar(int idx)
```

Arguments

`idx`: index of the desired variable.

Return

the desired variable object.

Model::GetVarByName()

Get a variable of given name in model.

Synopsis

```
Var GetVarByName(const char *szName)
```

Arguments

`szName`: name of the desired variable.

Return

the desired variable object.

Model::GetVarLowerIIS()

Get IIS status of lower bounds of variables.

Synopsis

```
int GetVarLowerIIS(const VarArray &vars, int *pLowerIIS)
```

Arguments

`vars`: Array of variables

`pLowerIIS`: IIS status of lower bounds of variables.

Return

Number of variables.

Model::GetVars()

Get all variables in model.

Synopsis

```
VarArray GetVars()
```

Return

variable array object.

Model::GetVarUpperIIS()

Get IIS status of upper bounds of variables.

Synopsis

```
int GetVarUpperIIS(const VarArray &vars, int *pUpperIIS)
```

Arguments

`vars`: Array of variables

`pUpperIIS`: IIS status of upper bounds of variables.

Return

Number of variables.

Model::Interrupt()

Interrupt optimization of current problem.

Synopsis

```
void Interrupt()
```

Model::LoadMipStart()

Load final initial values of variables to the problem.

Synopsis

```
void LoadMipStart()
```

Model::LoadTuneParam()

Load specified tuned parameters into model.

Synopsis

```
void LoadTuneParam(int idx)
```

Arguments

`idx`: Index of tuned parameters.

Model::Read()

Read problem, solution, basis, MIP start or COPT parameters from file.

Synopsis

```
void Read(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::ReadBasis()

Read basis from file.

Synopsis

```
void ReadBasis(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::ReadBin()

Read problem in COPT binary format from file.

Synopsis

```
void ReadBin(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::ReadCbf()

Read problem in CBF format from file.

Synopsis

```
void ReadCbf(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::ReadLp()

Read problem in LP format from file.

Synopsis

```
void ReadLp(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::ReadMps()

Read problem in MPS format from file.

Synopsis

```
void ReadMps(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::ReadMst()

Read MIP start information from file.

Synopsis

```
void ReadMst(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::ReadParam()

Read COPT parameters from file.

Synopsis

```
void ReadParam(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::ReadSdpa()

Read problem in SDPA format from file.

Synopsis

```
void ReadSdpa(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::ReadSol()

Read solution from file.

Synopsis

```
void ReadSol(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::ReadTune()

Read tuning parameters from file.

Synopsis

```
void ReadTune(const char *szFileName)
```

Arguments

`szFileName`: an input file name.

Model::Remove()

Remove a list of variables from model.

Synopsis

```
void Remove(VarArray &vars)
```

Arguments

`vars`: an array of variables.

Model::Remove()

Remove a list of constraints from model.

Synopsis

```
void Remove(ConstrArray &constrs)
```

Arguments

`constrs`: an array of constraints.

Model::Remove()

Remove a list of SOS constraints from model.

Synopsis

```
void Remove(SosArray &soss)
```

Arguments

`soss`: an array of SOS constraints.

Model::Remove()

Remove a list of gernal constraints from model.

Synopsis

```
void Remove(GenConstrArray &genConstrs)
```

Arguments

`genConstrs`: an array of general constraints.

Model::Remove()

Remove a list of Cone constraints from model.

Synopsis

```
void Remove(ConeArray &cones)
```

Arguments

`cones`: an array of Cone constraints.

Model::Remove()

Remove a list of quadratic constraints from model.

Synopsis

```
void Remove(QConstrArray &qconstrs)
```

Arguments

`qconstrs`: an array of quadratic constraints.

Model::Remove()

Remove a list of PSD variables from model.

Synopsis

```
void Remove(PsdVarArray &vars)
```

Arguments

`vars`: an array of PSD variables.

Model::Remove()

Remove a list of PSD constraints from model.

Synopsis

```
void Remove(PsdConstrArray &constrs)
```

Arguments

`constrs`: an array of PSD constraints.

Model::Reset()

Reset solution only.

Synopsis

```
void Reset()
```

Model::ResetAll()

Reset solution and additional information.

Synopsis

```
void ResetAll()
```

Model::ResetParam()

Reset parameters to default settings.

Synopsis

```
void ResetParam()
```

Model::Set()

Set values of double parameter, associated with variables.

Synopsis

```
void Set(
    const char *szName,
    const VarArray &vars,
    double *pVals,
    int len)
```

Arguments

`szName`: name of double parameter.

`vars`: a list of desired variables.

`pVals`: array of parameter values.

`len`: length of value array.

Model::Set()

Set values of parameter, associated with constraints.

Synopsis

```
void Set(
    const char *szName,
    const ConstrArray &constrs,
    double *pVals,
    int len)
```

Arguments

`szName`: name of double parameter.

`constrs`: a list of desired constraints.

`pVals`: array of parameter values.

`len`: length of value array.

Model::Set()

Set values of parameter, associated with PSD constraints.

Synopsis

```
void Set(
    const char *szName,
    const PsdConstrArray &constrs,
    double *pVals,
    int len)
```

Arguments

`szName`: name of double parameter.

`constrs`: a list of desired PSD constraints.

`pVals`: array of values of parameter.

`len`: length of value array.

Model::SetBasis()

Set column and row basis status to model.

Synopsis

```
void SetBasis(int *pColBasis, int *pRowBasis)
```

Arguments

`pColBasis`: pointer to status of column basis.

`pRowBasis`: pointer to status of row basis.

Model::SetCallback()

Set user callback to COPT model.

Synopsis

```
void SetCallback(ICallback *pcb, int cbctx)
```

Arguments

`pcb`: pointer to user callback object.

`cbctx`: COPT callback context, see definition in copt.h

Model::SetCoeff()

Set the coefficient of a variable in a linear constraint.

Synopsis

```
void SetCoeff(
    const Constraint &constr,
    const Var &var,
    double newVal)
```

Arguments

`constr`: The requested constraint.

`var`: The requested variable.

`newVal`: New coefficient.

Model::SetDblParam()

Set value of a COPT double parameter.

Synopsis

```
void SetDblParam(const char *szParam, double dVal)
```

Arguments

`szParam`: name of integer parameter.

`dVal`: double value.

Model::SetIntParam()

Set value of a COPT integer parameter.

Synopsis

```
void SetIntParam(const char *szParam, int nVal)
```

Arguments

`szParam`: name of integer parameter.

`nVal`: integer value.

Model::SetLpSolution()

Set LP solution.

Synopsis

```
void SetLpSolution(
    double *pValue,
    double *pSlack,
    double *pRowDual,
    double *pRedCost)
```

Arguments

`pValue`: double pointer to solution values.

`pSlack`: double poitner to slack values.

`pRowDual`: double pointer to dual values.

`pRedCost`: double pointer to reduced costs.

Model::SetMipStart()

Set initial values for variables of given number, starting from the first one.

Synopsis

```
void SetMipStart(int count, double *pVals)
```

Arguments

`count`: the number of variables to set.

`pVals`: pointer to initial values of variables.

Model::SetMipStart()

Set initial value for the specified variable.

Synopsis

```
void SetMipStart(const Var &var, double val)
```

Arguments

`var`: an interested variable.

`val`: initial value of the variable.

Model::SetMipStart()

Set initial values for an array of variables.

Synopsis

```
void SetMipStart(const VarArray &vars, double *pVals)
```

Arguments

`vars`: a list of interested variables.

`pVals`: pointer to initial values of variables.

Model::SetObjConst()

Set objective constant.

Synopsis

```
void SetObjConst(double constant)
```

Arguments

`constant`: constant value to set.

Model::SetObjective()

Set objective for model.

Synopsis

```
void SetObjective(const Expr &expr, int sense)
```

Arguments

`expr`: expression of the objective.

`sense`: optional, default value 0 does not change COPT sense.

Model::SetObjSense()

Set objective sense for model.

Synopsis

```
void SetObjSense(int sense)
```

Arguments

`sense`: the objective sense.

Model::SetPsdCoeff()

Set the coefficient matrix of a PSD variable in a PSD constraint.

Synopsis

```
void SetPsdCoeff(
    const PsdConstraint &constr,
    const PsdVar &var,
    const SymMatrix &mat)
```

Arguments

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

`mat`: new coefficient matrix.

Model::SetPsdObjective()

Set PSD objective for model.

Synopsis

```
void SetPsdObjective(const PsdExpr &expr, int sense)
```

Arguments

`expr`: PSD expression of the objective.

`sense`: optional, default value 0 does not change COPT sense.

Model::SetQuadObjective()

Set quadratic objective for model.

Synopsis

```
void SetQuadObjective(const QuadExpr &expr, int sense)
```

Arguments

`expr`: quadratic expression of the objective.

`sense`: optional, default value 0 does not change COPT sense.

Model::SetSlackBasis()

Set slack basis to model.

Synopsis

```
void SetSlackBasis()
```

Model::SetSolverLogCallback()

Set log callback for COPT.

Synopsis

```
void SetSolverLogCallback(ILogCallback *pcb)
```

Arguments

`pcb`: pointer to ILogCallback object.

Model::SetSolverLogFile()

Set log file for COPT.

Synopsis

```
void SetSolverLogFile(const char *szLogFile)
```

Arguments

`szLogFile`: log file name.

Model::Solve()

Solve the model as MIP.

Synopsis

```
void Solve()
```

Model::SolveLp()

Solve the model as LP.

Synopsis

```
void SolveLp()
```

Model::Tune()

Tune model.

Synopsis

```
void Tune()
```

Model::Write()

Output problem, solution, basis, MIP start or modified COPT parameters to file.

Synopsis

```
void Write(const char *szFileName)
```

Arguments

`szFileName`: an output file name.

Model::WriteBasis()

Output optimal basis to a file of type '.bas'.

Synopsis

```
void WriteBasis(const char *szFileName)
```

Arguments

`szFileName`: an output file name.

Model::WriteBin()

Output problem to a file as COPT binary format.

Synopsis

```
void WriteBin(const char *szFileName)
```

Arguments

`szFileName`: an output file name.

Model::WriteIIS()

Output IIS to file.

Synopsis

```
void WriteIIS(const char *szFileName)
```

Arguments

`szFileName`: Output file name.

Model::WriteLp()

Output problem to a file as LP format.

Synopsis

```
void WriteLp(const char *szFileName)
```

Arguments

`szFileName`: an output file name.

Model::WriteMps()

Output problem to a file as MPS format.

Synopsis

```
void WriteMps(const char *szFileName)
```

Arguments

`szFileName`: an output file name.

Model::WriteMpsStr()

Output problem to a buffer as MPS format.

Synopsis

```
ProbBuffer WriteMpsStr()
```

Return

output problem buffer.

Model::WriteMst()

Output MIP start information to a file of type '.mst'.

Synopsis

```
void WriteMst(const char *szFileName)
```

Arguments

`szFileName`: an output file name.

Model::WriteParam()

Output modified COPT parameters to a file of type '.par'.

Synopsis

```
void WriteParam(const char *szFileName)
```

Arguments

`szFileName`: an output file name.

Model::WritePoolSol()

Output selected pool solution to a file of type '.sol'.

Synopsis

```
void WritePoolSol(int iSol, const char *szFileName)
```

Arguments

`iSol`: index of pool solution.

`szFileName`: an output file name.

Model::WriteRelax()

Output feasibility relaxation problem to file.

Synopsis

```
void WriteRelax(const char *szFileName)
```

Arguments

`szFileName`: Output file name.

Model::WriteSol()

Output solution to a file of type '.sol'.

Synopsis

```
void WriteSol(const char *szFileName)
```

Arguments

`szFileName`: an output file name.

Model::WriteTuneParam()

Output specified tuned parameters to a file of type '.par'.

Synopsis

```
void WriteTuneParam(int idx, const char *szFileName)
```

Arguments

`idx`: Index of tuned parameters.

`szFileName`: Output file name.

18.4.4 Var

COPT variable object. Variables are always associated with a particular model. User creates a variable object by adding a variable to a model, rather than by using constructor of Var class.

Var::Get()

Get attribute value of the variable. Support "Value", "RedCost", "LB", "UB", and "Obj" attributes.

Synopsis

```
double Get(const char *szAttr)
```

Arguments

`szAttr`: attribute name.

Return

attribute value

Var::GetIdx()

Get index of the variable.

Synopsis

```
inline int GetIdx()
```

Return

variable index.

Var::GetLowerIIS()

Get IIS status for lower bound of the variable.

Synopsis

```
int GetLowerIIS()
```

Return

IIS status.

Var::GetName()

Get name of the variable.

Synopsis

```
const char *GetName()
```

Return

variable name.

Var::GetType()

Get type of the variable.

Synopsis

```
char GetType()
```

Return

variable type.

Var::GetUpperIIS()

Get IIS status for upper bound of the variable.

Synopsis

```
int GetUpperIIS()
```

Return

IIS status.

Var::Remove()

Remove variable from model.

Synopsis

```
void Remove()
```

Var::Set()

Set attribute value of the variable. Support "LB", "UB" and "Obj" attributes.

Synopsis

```
void Set(const char *szAttr, double value)
```

Arguments

`szAttr`: attribute name.

`value`: new value.

Var::SetName()

Set name of the variable.

Synopsis

```
void SetName(const char *szName)
```

Arguments

`szName`: variable name.

Var::SetType()

Set type of the variable.

Synopsis

```
void SetType(char type)
```

Arguments

`type`: variable type.

18.4.5 VarArray

COPT variable array object. To store and access a set of C++ *Var* objects, Cardinal Optimizer provides C++ VarArray class, which defines the following methods.

VarArray::GetVar()

Get i-th variable object.

Synopsis

```
Var &GetVar(int i)
```

Arguments

`i`: index of the variable.

Return

variable object with index i.

VarArray::PushBack()

Add a variable object to variable array.

Synopsis

```
void PushBack(const Var &var)
```

Arguments

`var`: a variable object.

VarArray::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for variable object.

VarArray::Size()

Get the number of variable objects.

Synopsis

```
int Size()
```

Return

number of variable objects.

18.4.6 Expr

COPT linear expression object. A linear expression consists of a constant term, a list of terms of variables and associated coefficients. Linear expressions are used to build constraints.

Expr::Expr()

Constructor of a constant linear expression.

Synopsis

```
Expr(double constant)
```

Arguments

`constant`: constant value in expression object.

Expr::Expr()

Constructor of a linear expression with one term.

Synopsis

```
Expr(const Var &var, double coeff)
```

Arguments

`var`: variable for the added term.

`coeff`: coefficent for the added term.

Expr::AddConstant()

Add constant for the expression.

Synopsis

```
void AddConstant(double constant)
```

Arguments

`constant`: the value of the constant.

Expr::AddExpr()

Add an expression to self.

Synopsis

```
void AddExpr(const Expr &expr, double mult)
```

Arguments

`expr`: expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

Expr::AddTerm()

Add a term to expression object.

Synopsis

```
void AddTerm(const Var &var, double coeff)
```

Arguments

`var`: a variable for new term.

`coeff`: coefficient for new term.

Expr::AddTerms()

Add terms to expression object.

Synopsis

```
int AddTerms(
    const VarArray &vars,
    double *pCoeff,
    int len)
```

Arguments

`vars`: variables for added terms.

`pCoeff`: coefficient array for added terms.

`len`: length of coefficient array.

Return

number of added terms.

Expr::Clone()

Deep copy linear expression object.

Synopsis

```
Expr Clone()
```

Return

cloned expression object.

Expr::Evaluate()

evaluate linear expression after solving

Synopsis

```
double Evaluate()
```

Return

value of linear expression

Expr::GetCoeff()

Get coefficient from the i-th term in expression.

Synopsis

```
double GetCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term in expression object.

Expr::GetConstant()

Get constant in expression.

Synopsis

```
double GetConstant()
```

Return

constant in expression.

Expr::GetVar()

Get variable from the i-th term in expression.

Synopsis

```
Var &GetVar(int i)
```

Arguments

`i`: index of the term.

Return

variable of the i-th term in expression object.

Expr::operator*=()

Multiply a constant to self.

Synopsis

```
void operator*=(double c)
```

Arguments

`c`: constant multiplier.

Expr::operator*()

Multiply constant and return new expression.

Synopsis

```
Expr operator*(double c)
```

Arguments

`c`: constant multiplier.

Return

result expression.

Expr::operator*()

Multiply a variable and return new quadratic expression object.

Synopsis

```
QuadExpr operator*(const Var &var)
```

Arguments

`var`: variable object.

Return

result quadratic expression.

Expr::operator*()

Multiply a linear expression and return new quadratic expression object.

Synopsis

```
QuadExpr operator*(const Expr &other)
```

Arguments

`other`: linear expression object.

Return

result quadratic expression.

Expr::operator+=()

Add an expression to self.

Synopsis

```
void operator+=(const Expr &expr)
```

Arguments

`expr`: expression to be added.

Expr::operator+()

Add expression and return new expression.

Synopsis

```
Expr operator+(const Expr &other)
```

Arguments

`other`: other expression to add.

Return

result expression.

Expr::operator-=()

Substract an expression from self.

Synopsis

```
void operator-=(const Expr &expr)
```

Arguments

`expr`: expression to be substracted.

Expr::operator-()

Substract expression and return new expression.

Synopsis

```
Expr operator-(const Expr &other)
```

Arguments

`other`: other expression to substract.

Return

result expression.

Expr::Remove()

Remove i-th term from expression object.

Synopsis

```
void Remove(int i)
```

Arguments

`i`: index of the term to be removed.

Expr::Remove()

Remove the term associated with variable from expression.

Synopsis

```
void Remove(const Var &var)
```

Arguments

`var`: a variable whose term should be removed.

Expr::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(size_t n)
```

Arguments

`n`: minimum capacity for linear expression object.

Expr::SetCoeff()

Set coefficient for the i-th term in expression.

Synopsis

```
void SetCoeff(int i, double val)
```

Arguments

`i`: index of the term.

`val`: coefficient of the term.

Expr::SetConstant()

Set constant for the expression.

Synopsis

```
void SetConstant(double constant)
```

Arguments

`constant`: the value of the constant.

Expr::Size()

Get number of terms in expression.

Synopsis

```
size_t Size()
```

Return

number of terms.

18.4.7 Constraint

COPT constraint object. Constraints are always associated with a particular model. User creates a constraint object by adding a constraint to a model, rather than by using constructor of Constraint class.

Constraint::Get()

Get attribute value of the constraint. Support "Dual", "Slack", "LB", "UB" attributes.

Synopsis

```
double Get(const char *szAttr)
```

Arguments

`szAttr`: name of the attribute being queried.

Return

attribute value.

Constraint::GetBasis()

Get basis status of this constraint.

Synopsis

```
int GetBasis()
```

Return

basis status.

Constraint::GetIdx()

Get index of the constraint.

Synopsis

```
int GetIdx()
```

Return

the index of the constraint.

Constraint::GetLowerIIS()

Get IIS status for lower bound of the constraint.

Synopsis

```
int GetLowerIIS()
```

Return

IIS status.

Constraint::GetName()

Get name of the constraint.

Synopsis

```
const char *GetName()
```

Return

the name of the constraint.

Constraint::GetUpperIIS()

Get IIS status for upper bound of the constraint.

Synopsis

```
int GetUpperIIS()
```

Return

IIS status.

Constraint::Remove()

Remove this constraint from model.

Synopsis

```
void Remove()
```

Constraint::Set()

Set attribute value of the constraint. Support "LB" and "UB" attributes.

Synopsis

```
void Set(const char *szAttr, double value)
```

Arguments

`szAttr`: name of the attribute.

`value`: new value.

Constraint::SetName()

Set name for the constraint.

Synopsis

```
void SetName(const char *szName)
```

Arguments

`szName`: the name to set.

18.4.8 ConstrArray

COPT constraint array object. To store and access a set of C++ *Constraint* objects, Cardinal Optimizer provides C++ ConstrArray class, which defines the following methods.

ConstrArray::GetConstr()

Get i-th constraint object.

Synopsis

```
Constraint &GetConstr(int i)
```

Arguments

`i`: index of the constraint.

Return

constraint object with index i.

ConstrArray::PushBack()

Add a constraint object to constraint array.

Synopsis

```
void PushBack(const Constraint &constr)
```

Arguments

`constr`: a constraint object.

ConstrArray::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for Constraint object.

ConstrArray::Size()

Get the number of constraint objects.

Synopsis

```
int Size()
```

Return

number of constraint objects.

18.4.9 ConstrBuilder

COPT constraint builder object. To help building a constraint, given a linear expression, constraint sense and right-hand side value, Cardinal Optimizer provides C++ ConstrBuilder class, which defines the following methods.

ConstrBuilder::GetExpr()

Get expression associated with constraint.

Synopsis

```
const Expr &GetExpr()
```

Return

expression object.

ConstrBuilder::GetRange()

Get range from lower bound to upper bound of range constraint.

Synopsis

```
double GetRange()
```

Return

length from lower bound to upper bound of the constraint.

ConstrBuilder::GetSense()

Get sense associated with constraint.

Synopsis

```
char GetSense()
```

Return

constraint sense.

ConstrBuilder::Set()

Set detail of a constraint to its builder object.

Synopsis

```
void Set(
    const Expr &expr,
    char sense,
    double rhs)
```

Arguments

`expr`: expression object at one side of the constraint

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of the constraint.

ConstrBuilder::SetRange()

Set a range constraint to its builder.

Synopsis

```
void SetRange(const Expr &expr, double range)
```

Arguments

`expr`: expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must greater than 0.

18.4.10 ConstrBuilderArray

COPT constraint builder array object. To store and access a set of C++ *ConstrBuilder* objects, Cardinal Optimizer provides C++ ConstrBuilderArray class, which defines the following methods.

ConstrBuilderArray::GetBuilder()

Get i-th constraint builder object.

Synopsis

```
ConstrBuilder &GetBuilder(int i)
```

Arguments

`i`: index of the constraint builder.

Return

constraint builder object with index i.

ConstrBuilderArray::PushBack()

Add a constraint builder object to constraint builder array.

Synopsis

```
void PushBack(const ConstrBuilder &builder)
```

Arguments

`builder`: a constraint builder object.

ConstrBuilderArray::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for constraint builder object.

ConstrBuilderArray::Size()

Get the number of constraint builder objects.

Synopsis

```
int Size()
```

Return

number of constraint builder objects.

18.4.11 Column

COPT column object. A column consists of a list of constraints and associated coefficients. Columns are used to represent the set of constraints in which a variable participates, and the asssociated coefficents.

Column::Column()

Constructor of column.

Synopsis

```
Column()
```

Column::AddColumn()

Add a column to self.

Synopsis

```
void AddColumn(const Column &col, double mult)
```

Arguments

`col`: column object to be added.

`mult`: multiply constant.

Column::AddTerm()

Add a term to column object.

Synopsis

```
void AddTerm(const Constraint &constr, double coeff)
```

Arguments

`constr`: a constraint for new term.

`coeff`: coefficient for new term.

Column::AddTerms()

Add terms to column object.

Synopsis

```
int AddTerms(
    const ConstrArray &constrs,
    double *pCoeff,
    int len)
```

Arguments

`constrs`: constraints for added terms.

`pCoeff`: coefficients for added terms.

`len`: number of terms to be added.

Return

number of added terms.

Column::Clear()

Clear all terms.

Synopsis

```
void Clear()
```

Column::Clone()

Deep copy column object.

Synopsis

```
Column Clone()
```

Return

cloned column object.

Column::GetCoeff()

Get coefficient from the i-th term in column object.

Synopsis

```
double GetCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term in column object.

Column::GetConstr()

Get constraint from the i-th term in column object.

Synopsis

```
Constraint GetConstr(int i)
```

Arguments

`i`: index of the term.

Return

constraint of the i-th term in column object.

Column::Remove()

Remove i-th term from column object.

Synopsis

```
void Remove(int i)
```

Arguments

`i`: index of the term to be removed.

Column::Remove()

Remove the term associated with constraint from column object.

Synopsis

```
bool Remove(const Constraint &constr)
```

Arguments

`constr`: a constraint whose term should be removed.

Return

true if constraint exits in column object.

Column::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for Column object.

Column::Size()

Get number of terms in column object.

Synopsis

```
int Size()
```

Return

number of terms.

18.4.12 ColumnArray

COPT column array object. To store and access a set of C++ *Column* objects, Cardinal Optimizer provides C++ ColumnArray class, which defines the following methods.

ColumnArray::Clear()

Clear all column objects.

Synopsis

```
void Clear()
```

ColumnArray::GetColumn()

Get i-th column object.

Synopsis

```
Column &GetColumn(int i)
```

Arguments

`i`: index of the column.

Return

column object with index i.

ColumnArray::PushBack()

Add a column object to column array.

Synopsis

```
void PushBack(const Column &col)
```

Arguments

`col`: a column object.

ColumnArray::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for linear expression object.

ColumnArray::Size()

Get the number of column objects.

Synopsis

```
int Size()
```

Return

number of column objects.

18.4.13 Sos

COPT SOS constraint object. SOS constraints are always associated with a particular model. User creates an SOS constraint object by adding an SOS constraint to a model, rather than by using constructor of Sos class.

An SOS constraint can be type 1 or 2 (`COPT_SOS_TYPE1` or `COPT_SOS_TYPE2`).

Sos::GetIdx()

Get the index of SOS constraint.

Synopsis

```
int GetIdx()
```

Return

index of SOS constraint.

Sos::GetIIS()

Get IIS status of the SOS constraint.

Synopsis

```
int GetIIS()
```

Return

IIS status.

Sos::Remove()

Remove the SOS constraint from model.

Synopsis

```
void Remove()
```

18.4.14 SosArray

COPT SOS constraint array object. To store and access a set of C++ *Sos* objects, Cardinal Optimizer provides C++ SosArray class, which defines the following methods.

SosArray::GetSos()

Get i-th SOS constraint object.

Synopsis

```
Sos &GetSos(int i)
```

Arguments

`i`: index of the SOS constraint.

Return

SOS constraint object with index i.

SosArray::PushBack()

Add a SOS constraint object to SOS constraint array.

Synopsis

```
void PushBack(const Sos &sos)
```

Arguments

`sos`: a SOS constraint object.

SosArray::Size()

Get the number of SOS constraint objects.

Synopsis

```
int Size()
```

Return

number of SOS constraint objects.

18.4.15 SosBuilder

COPT SOS constraint builder object. To help building an SOS constraint, given the SOS type, a set of variables and associated weights, Cardinal Optimizer provides C++ SosBuilder class, which defines the following methods.

SosBuilder::GetSize()

Get number of terms in SOS constraint.

Synopsis

```
int GetSize()
```

Return

number of terms.

SosBuilder::GetType()

Get type of SOS constraint.

Synopsis

```
int GetType()
```

Return

type of SOS constraint.

SosBuilder::GetVar()

Get variable from the i-th term in SOS constraint.

Synopsis

```
Var GetVar(int i)
```

Arguments

`i`: index of the term.

Return

variable of the i-th term in SOS constraint.

SosBuilder::GetVars()

Get variables of all terms in SOS constraint.

Synopsis

```
VarArray GetVars()
```

Return

variable array object.

SosBuilder::GetWeight()

Get weight from the i-th term in SOS constraint.

Synopsis

```
double GetWeight(int i)
```

Arguments

`i`: index of the term.

Return

weight of the i-th term in SOS constraint.

SosBuilder::GetWeights()

Get weights of all terms in SOS constraint.

Synopsis

```
double GetWeights()
```

Return

pointer to array of weights.

SosBuilder::Set()

Set variables and weights of SOS constraint.

Synopsis

```
void Set(
    const VarArray &vars,
    const double *pWeights,
    int type)
```

Arguments

`vars`: variable array object.

`pWeights`: pointer to array of weights.

`type`: type of SOS constraint.

18.4.16 SosBuilderArray

COPT SOS constraint builder array object. To store and access a set of C++ *SosBuilder* objects, Cardinal Optimizer provides C++ SosBuilderArray class, which defines the following methods.

SosBuilderArray::GetBuilder()

Get i-th SOS constraint builder object.

Synopsis

```
SosBuilder &GetBuilder(int i)
```

Arguments

`i`: index of the SOS constraint builder.

Return

SOS constraint builder object with index i.

SosBuilderArray::PushBack()

Add a SOS constraint builder object to SOS constraint builder array.

Synopsis

```
void PushBack(const SosBuilder &builder)
```

Arguments

`builder`: a SOS constraint builder object.

SosBuilderArray::Size()

Get the number of SOS constraint builder objects.

Synopsis

```
int Size()
```

Return

number of SOS constraint builder objects.

18.4.17 GenConstr

COPT general constraint object. General constraints are always associated with a particular model. User creates a general constraint object by adding a general constraint to a model, rather than by using constructor of GenConstr class.

GenConstr::GetIdx()

Get the index of the general constraint.

Synopsis

```
int GetIdx()
```

Return

index of the general constraint.

GenConstr::GetIIS()

Get IIS status of the general constraint.

Synopsis

```
int GetIIS()
```

Return

IIS status.

GenConstr::Remove()

Remove the general constraint from model.

Synopsis

```
void Remove()
```

18.4.18 GenConstrArray

COPT general constraint array object. To store and access a set of C++ *GenConstr* objects, Cardinal Optimizer provides C++ GenConstrArray class, which defines the following methods.

GenConstrArray::GetGenConstr()

Get i-th general constraint object.

Synopsis

```
GenConstr &GetGenConstr(int i)
```

Arguments

`i`: index of the general constraint.

Return

general constraint object with index i.

GenConstrArray::PushBack()

Add a general constraint object to general constraint array.

Synopsis

```
void PushBack(const GenConstr &constr)
```

Arguments

`constr`: a general constraint object.

GenConstrArray::Size()

Get the number of general constraint objects.

Synopsis

```
int Size()
```

Return

number of general constraint objects.

18.4.19 GenConstrBuilder

COPT general constraint builder object. To help building a general constraint, given a binary variable and associated value, a linear expression and constraint sense, Cardinal Optimizer provides C++ GenConstrBuilder class, which defines the following methods.

GenConstrBuilder::GetBinVal()

Get binary value associated with general constraint.

Synopsis

```
int GetBinVal()
```

Return

binary value.

GenConstrBuilder::GetBinVar()

Get binary variable associated with general constraint.

Synopsis

```
Var GetBinVar()
```

Return

binary vaiable object.

GenConstrBuilder::GetExpr()

Get expression associated with general constraint.

Synopsis

```
const Expr &GetExpr()
```

Return

expression object.

GenConstrBuilder::GetSense()

Get sense associated with general constraint.

Synopsis

```
char GetSense()
```

Return

constraint sense.

GenConstrBuilder::Set()

Set binary variable, binary value, expression and sense of general constraint.

Synopsis

```
void Set(
    Var var,
    int val,
    const Expr &expr,
    char sense)
```

Arguments

`var`: binary variable.

`val`: binary value.

`expr`: expression object.

`sense`: general constraint sense.

18.4.20 GenConstrBuilderArray

COPT general constraint builder array object. To store and access a set of C++ *GenConstrBuilder* objects, Cardinal Optimizer provides C++ GenConstrBuilderArray class, which defines the following methods.

GenConstrBuilderArray::GetBuilder()

Get i-th general constraint builder object.

Synopsis

```
GenConstrBuilder &GetBuilder(int i)
```

Arguments

`i`: index of the general constraint builder.

Return

general constraint builder object with index i.

GenConstrBuilderArray::PushBack()

Add a general constraint builder object to general constraint builder array.

Synopsis

```
void PushBack(const GenConstrBuilder &builder)
```

Arguments

`builder`: a general constraint builder object.

GenConstrBuilderArray::Size()

Get the number of general constraint builder objects.

Synopsis

```
int Size()
```

Return

number of general constraint builder objects.

18.4.21 Cone

COPT cone constraint object. Cone constraints are always associated with a particular model. User creates a cone constraint object by adding a cone constraint to a model, rather than by using constructor of Cone class.

A cone constraint can be regular or rotated (`COPT_CONE_QUAD` or `COPT_CONE_RQUAD`).

Cone::GetIdx()

Get the index of a cone constraint.

Synopsis

```
int GetIdx()
```

Return

index of a cone constraint.

Cone::Remove()

Remove the cone constraint from model.

Synopsis

```
void Remove()
```

18.4.22 ConeArray

COPT cone constraint array object. To store and access a set of C++ *Cone* objects, Cardinal Optimizer provides C++ ConeArray class, which defines the following methods.

ConeArray::GetCone()

Get i-th cone constraint object.

Synopsis

```
Cone &GetCone(int i)
```

Arguments

`i`: index of the cone constraint.

Return

cone constraint object with index i.

ConeArray::PushBack()

Add a cone constraint object to cone constraint array.

Synopsis

```
void PushBack(const Cone &cone)
```

Arguments

`cone`: a cone constraint object.

ConeArray::Size()

Get the number of cone constraint objects.

Synopsis

```
int Size()
```

Return

number of cone constraint objects.

18.4.23 ConeBuilder

COPT cone constraint builder object. To help building a cone constraint, given the cone type and a set of variables, Cardinal Optimizer provides C++ ConeBuilder class, which defines the following methods.

ConeBuilder::GetSize()

Get number of vars in a cone constraint.

Synopsis

```
int GetSize()
```

Return

number of vars.

ConeBuilder::GetType()

Get type of a cone constraint.

Synopsis

```
int GetType()
```

Return

type of a cone constraint.

ConeBuilder::GetVar()

Get the i-th variable in a cone constraint.

Synopsis

```
Var GetVar(int i)
```

Arguments

`i`: index of vars in a cone constraint.

Return

the i-th variable in a cone constraint.

ConeBuilder::GetVars()

Get all variables in a cone constraint.

Synopsis

```
VarArray GetVars()
```

Return

variable array object.

ConeBuilder::Set()

Set variables of a cone constraint.

Synopsis

```
void Set(const VarArray &vars, int type)
```

Arguments

`vars`: variable array object.

`type`: type of cone constraint.

18.4.24 ConeBuilderArray

COPT cone constraint builder array object. To store and access a set of C++ *ConeBuilder* objects, Cardinal Optimizer provides C++ ConeBuilderArray class, which defines the following methods.

ConeBuilderArray::GetBuilder()

Get i-th cone constraint builder object.

Synopsis

```
ConeBuilder &GetBuilder(int i)
```

Arguments

`i`: index of the cone constraint builder.

Return

cone constraint builder object with index i.

ConeBuilderArray::PushBack()

Add a cone constraint builder object to cone constraint builder array.

Synopsis

```
void PushBack(const ConeBuilder &builder)
```

Arguments

`builder`: a cone constraint builder object.

ConeBuilderArray::Size()

Get the number of cone constraint builder objects.

Synopsis

```
int Size()
```

Return

number of cone constraint builder objects.

18.4.25 QuadExpr

COPT quadratic expression object. A quadratic expression consists of a linear expression, a list of variable pairs and associated coefficients of quadratic terms. Quadratic expressions are used to build quadratic constraints and objectives.

QuadExpr::QuadExpr()

Constructor of a quadratic expression with a constant.

Synopsis

```
QuadExpr(double constant)
```

Arguments

`constant`: constant value in quadratic expression object.

QuadExpr::QuadExpr()

Constructor of a quadratic expression with one term.

Synopsis

```
QuadExpr(const Var &var, double coeff)
```

Arguments

`var`: variable for the added term.

`coeff`: coefficent for the added term.

QuadExpr::QuadExpr()

Constructor of a quadratic expression with a linear expression.

Synopsis

```
QuadExpr(const Expr &expr)
```

Arguments

`expr`: input linear expression.

QuadExpr::QuadExpr()

Constructor of a quadratic expression with two linear expression.

Synopsis

```
QuadExpr(const Expr &expr, const Var &var)
```

Arguments

`expr`: one linear expression.

`var`: another variable.

QuadExpr::QuadExpr()

Constructor of a quadratic expression with two linear expression.

Synopsis

```
QuadExpr(const Expr &left, const Expr &right)
```

Arguments

`left`: one linear expression.

`right`: another linear expression.

QuadExpr::AddConstant()

Add constant for the expression.

Synopsis

```
void AddConstant(double constant)
```

Arguments

`constant`: the value of the constant.

QuadExpr::AddLinExpr()

Add a linear expression to self.

Synopsis

```
void AddLinExpr(const Expr &expr, double mult)
```

Arguments

`expr`: linear expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

QuadExpr::AddQuadExpr()

Add a quadratic expression to self.

Synopsis

```
void AddQuadExpr(const QuadExpr &expr, double mult)
```

Arguments

`expr`: quadratic expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

QuadExpr::AddTerm()

Add a linear term to expression object.

Synopsis

```
void AddTerm(const Var &var, double coeff)
```

Arguments

`var`: variable of new linear term.

`coeff`: coefficient of new linear term.

QuadExpr::AddTerm()

Add a quadratic term to expression object.

Synopsis

```
void AddTerm(
    const Var &var1,
    const Var &var2,
    double coeff)
```

Arguments

`var1`: first variable of new quadratic term.

`var2`: second variable of new quadratic term.

`coeff`: coefficient of new quadratic term.

QuadExpr::AddTerms()

Add linear terms to expression object.

Synopsis

```
int AddTerms(
    const VarArray &vars,
    double *pCoeff,
    int len)
```

Arguments

`vars`: variables for added linear terms.

`pCoeff`: coefficient array for added linear terms.

`len`: length of coefficient array.

Return

number of added linear terms.

QuadExpr::AddTerms()

Add quadratic terms to expression object.

Synopsis

```
int AddTerms(
    const VarArray &vars1,
    const VarArray &vars2,
    double *pCoeff,
    int len)
```

Arguments

`vars1`: first set of variables for added quadratic terms.

`vars2`: second set of variables for added quadratic terms.

`pCoeff`: coefficient array for added quadratic terms.

`len`: length of coefficient array.

Return

number of added quadratic terms.

QuadExpr::Clone()

Deep copy quadratic expression object.

Synopsis

```
QuadExpr Clone()
```

Return

cloned quadratic expression object.

QuadExpr::Evaluate()

evaluate quadratic expression after solving

Synopsis

```
double Evaluate()
```

Return

value of quadratic expression

QuadExpr::GetCoeff()

Get coefficient from the i-th term in quadratic expression.

Synopsis

```
double GetCoeff(int i)
```

Arguments

`i`: index of the quadratic term.

Return

coefficient of the i-th quadratic term in quadratic expression object.

QuadExpr::GetConstant()

Get constant in quadratic expression.

Synopsis

```
double GetConstant()
```

Return

constant in quadratic expression.

QuadExpr::GetLinExpr()

Get linear expression in quadratic expression.

Synopsis

```
Expr &GetLinExpr()
```

Return

linear expression object.

QuadExpr::GetVar1()

Get the first variable from the i-th term in quadratic expression.

Synopsis

```
Var &GetVar1(int i)
```

Arguments

`i`: index of the term.

Return

the first variable of the i-th term in quadratic expression object.

QuadExpr::GetVar2()

Get the second variable from the i-th term in quadratic expression.

Synopsis

```
Var &GetVar2(int i)
```

Arguments

`i`: index of the term.

Return

the second variable of the i-th term in quadratic expression object.

QuadExpr::operator*=()

Multiply a constant to self.

Synopsis

```
void operator*=(double c)
```

Arguments

`c`: constant multiplier.

QuadExpr::operator*()

Multiply constant and return new expression.

Synopsis

```
QuadExpr operator*(double c)
```

Arguments

`c`: constant multiplier.

Return

result expression.

QuadExpr::operator+=()

Add an expression to self.

Synopsis

```
void operator+=(const QuadExpr &expr)
```

Arguments

`expr`: expression to be added.

QuadExpr::operator+()

Add expression and return new expression.

Synopsis

```
QuadExpr operator+(const QuadExpr &other)
```

Arguments

`other`: other expression to add.

Return

result expression.

QuadExpr::operator-=()

Substract an expression from self.

Synopsis

```
void operator-=(const QuadExpr &expr)
```

Arguments

`expr`: expression to be substracted.

QuadExpr::operator-()

Substract expression and return new expression.

Synopsis

```
QuadExpr operator-(const QuadExpr &other)
```

Arguments

`other`: other expression to substract.

Return

result expression.

QuadExpr::Remove()

Remove i-th term from expression object.

Synopsis

```
void Remove(int i)
```

Arguments

`i`: index of the term to be removed.

QuadExpr::Remove()

Remove the term associated with variable from expression.

Synopsis

```
void Remove(const Var &var)
```

Arguments

`var`: a variable whose term should be removed.

QuadExpr::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(size_t n)
```

Arguments

`n`: minimum capacity for quadratic expression object.

QuadExpr::SetCoeff()

Set coefficient of the i-th term in quadratic expression.

Synopsis

```
void SetCoeff(int i, double val)
```

Arguments

`i`: index of the quadratic term.

`val`: coefficient of the term.

QuadExpr::SetConstant()

Set constant for the expression.

Synopsis

```
void SetConstant(double constant)
```

Arguments

`constant`: the value of the constant.

QuadExpr::Size()

Get number of terms in expression.

Synopsis

```
size_t Size()
```

Return

number of terms.

18.4.26 QConstraint

COPT quadratic constraint object. Quadratic constraints are always associated with a particular model. User creates a quadratic constraint object by adding a quadratic constraint to a model, rather than by using constructor of QConstraint class.

QConstraint::Get()

Get attribute value of the quadratic constraint. Support related quadratic attributes.

Synopsis

```
double Get(const char *szAttr)
```

Arguments

`szAttr`: name of the attribute being queried.

Return

attribute value.

QConstraint::GetIdx()

Get index of the quadratic constraint.

Synopsis

```
int GetIdx()
```

Return

the index of the quadratic constraint.

QConstraint::GetName()

Get name of the quadratic constraint.

Synopsis

```
const char *GetName()
```

Return

the name of the quadratic constraint.

QConstraint::GetRhs()

Get rhs of quadratic constraint.

Synopsis

```
double GetRhs()
```

Return

rhs of quadratic constraint.

QConstraint::GetSense()

Get sense of quadratic constraint.

Synopsis

```
char GetSense()
```

Return

sense of quadratic constraint.

QConstraint::Remove()

Remove this quadratic constraint from model.

Synopsis

```
void Remove()
```

QConstraint::Set()

Set attribute value of the quadratic constraint. Support related quadratic attributes.

Synopsis

```
void Set(const char *szAttr, double value)
```

Arguments

`szAttr`: name of the attribute.

`value`: new value.

QConstraint::SetName()

Set name of a quadratic constraint.

Synopsis

```
void SetName(const char *szName)
```

Arguments

`szName`: the name to set.

QConstraint::SetRhs()

Set rhs of quadratic constraint.

Synopsis

```
void SetRhs(double rhs)
```

Arguments

`rhs`: rhs of quadratic constraint.

QConstraint::SetSense()

Set sense of quadratic constraint.

Synopsis

```
void SetSense(char sense)
```

Arguments

`sense`: sense of quadratic constraint.

18.4.27 QConstrArray

COPT quadratic constraint array object. To store and access a set of C++ *QConstraint* objects, Cardinal Optimizer provides C++ QConstrArray class, which defines the following methods.

QConstrArray::GetQConstr()

Get i-th quadratic constraint object.

Synopsis

```
QConstraint &GetQConstr(int idx)
```

Arguments

`idx`: index of the quadratic constraint.

Return

quadratic constraint object with index i.

QConstrArray::PushBack()

Add a quadratic constraint object to constraint array.

Synopsis

```
void PushBack(const QConstraint &constr)
```

Arguments

`constr`: a quadratic constraint object.

QConstrArray::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for quadratic constraint objects.

QConstrArray::Size()

Get the number of quadratic constraint objects.

Synopsis

```
int Size()
```

Return

number of quadratic constraint objects.

18.4.28 QConstrBuilder

COPT quadratic constraint builder object. To help building a quadratic constraint, given a quadratic expression, constraint sense and right-hand side value, Cardinal Optimizer provides C++ QConstrBuilder class, which defines the following methods.

QConstrBuilder::GetQuadExpr()

Get expression associated with quadratic constraint.

Synopsis

```
const QuadExpr &GetQuadExpr()
```

Return

quadratic expression object.

QConstrBuilder::GetSense()

Get sense associated with quadratic constraint.

Synopsis

```
char GetSense()
```

Return

quadratic constraint sense.

QConstrBuilder::Set()

Set detail of a quadratic constraint to its builder object.

Synopsis

```
void Set(
    const QuadExpr &expr,
    char sense,
    double rhs)
```

Arguments

`expr`: expression object at one side of the quadratic constraint.

`sense`: quadratic constraint sense.

`rhs`: constant of right side of quadratic constraint.

18.4.29 QConstrBuilderArray

COPT quadratic constraint builder array object. To store and access a set of C++ *QConstrBuilder* objects, Cardinal Optimizer provides C++ QConstrBuilderArray class, which defines the following methods.

QConstrBuilderArray::GetBuilder()

Get i-th quadratic constraint builder object.

Synopsis

```
QConstrBuilder &GetBuilder(int idx)
```

Arguments

`idx`: index of the quadratic constraint builder.

Return

quadratic constraint builder object with index i.

QConstrBuilderArray::PushBack()

Add a quadratic constraint builder object to quadratic constraint builder array.

Synopsis

```
void PushBack(const QConstrBuilder &builder)
```

Arguments

`builder`: a quadratic constraint builder object.

QConstrBuilderArray::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for quadratic constraint builder object.

QConstrBuilderArray::Size()

Get the number of quadratic constraint builder objects.

Synopsis

```
int Size()
```

Return

number of quadratic constraint builder objects.

18.4.30 PsdVar

COPT PSD variable object. PSD variables are always associated with a particular model. User creates a PSD variable object by adding a PSD variable to model, rather than by constructor of PsdVar class.

PsdVar::Get()

Get attribute values of PSD variable.

Synopsis

```
double Get(const char *szAttr, int sz)
```

Arguments

`szAttr`: attribute name.

`sz`: length of the output array.

Return

output array of attribute values.

PsdVar::GetDim()

Get dimension of PSD variable.

Synopsis

```
int GetDim()
```

Return

dimension of PSD variable.

PsdVar::GetIdx()

Get index of PSD variable.

Synopsis

```
int GetIdx()
```

Return

index of PSD variable.

PsdVar::GetLen()

Get length of PSD variable.

Synopsis

```
int GetLen()
```

Return

length of PSD variable.

PsdVar::GetName()

Get name of PSD variable.

Synopsis

```
const char *GetName()
```

Return

name of PSD variable.

PsdVar::Remove()

Remove PSD variable from model.

Synopsis

```
void Remove()
```

PsdVar::SetName()

Set name of PSD variable.

Synopsis

```
void SetName(const char *szName)
```

Arguments

`szName`: name of PSD variable.

18.4.31 PsdVarArray

COPT PSD variable array object. To store and access a set of *PsdVar* objects, Cardinal Optimizer provides PsdVarArray class, which defines the following methods.

PsdVarArray::GetPsdVar()

Get idx-th PSD variable object.

Synopsis

```
PsdVar &GetPsdVar(int idx)
```

Arguments

`idx`: index of the PSD variable.

Return

PSD variable object with index idx.

PsdVarArray::PushBack()

Add a PSD variable object to PSD variable array.

Synopsis

```
void PushBack(const PsdVar &var)
```

Arguments

`var`: a PSD variable object.

PsdVarArray::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for PSD variable object.

PsdVarArray::Size()

Get the number of PSD variable objects.

Synopsis

```
int Size()
```

Return

number of PSD variable objects.

18.4.32 PsdExpr

COPT PSD expression object. A PSD expression consists of a linear expression, a list of PSD variables and associated coefficient matrices of PSD terms. PSD expressions are used to build PSD constraints and objectives.

PsdExpr::PsdExpr()

Constructor of a PSD expression with a constant.

Synopsis

```
PsdExpr(double constant)
```

Arguments

`constant`: constant value in PSD expression object.

PsdExpr::PsdExpr()

Constructor of a PSD expression with one term.

Synopsis

```
PsdExpr(const Var &var, double coeff)
```

Arguments

`var`: variable for the added term.

`coeff`: coefficent for the added term.

PsdExpr::PsdExpr()

Constructor of a PSD expression with a linear expression.

Synopsis

```
PsdExpr(const Expr &expr)
```

Arguments

`expr`: input linear expression.

PsdExpr::PsdExpr()

Constructor of a PSD expression with one term.

Synopsis

```
PsdExpr(const PsdVar &var, const SymMatrix &mat)
```

Arguments

`var`: PSD variable for the added term.

`mat`: coefficient matrix for the added term.

PsdExpr::PsdExpr()

Constructor of a PSD expression with one term.

Synopsis

```
PsdExpr(const PsdVar &var, const SymMatExpr &expr)
```

Arguments

`var`: PSD variable for the added term.

`expr`: coefficient expression of symmetric matrices for the added term.

PsdExpr::AddConstant()

Add constant to the PSD expression.

Synopsis

```
void AddConstant(double constant)
```

Arguments

`constant`: value to be added.

PsdExpr::AddLinExpr()

Add a linear expression to self.

Synopsis

```
void AddLinExpr(const Expr &expr, double mult)
```

Arguments

`expr`: linear expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

PsdExpr::AddPsdExpr()

Add a PSD expression to self.

Synopsis

```
void AddPsdExpr(const PsdExpr &expr, double mult)
```

Arguments

`expr`: PSD expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

PsdExpr::AddTerm()

Add a linear term to PSD expression object.

Synopsis

```
void AddTerm(const Var &var, double coeff)
```

Arguments

`var`: variable of new linear term.

`coeff`: coefficient of new linear term.

PsdExpr::AddTerm()

Add a PSD term to PSD expression object.

Synopsis

```
void AddTerm(const PsdVar &var, const SymMatrix &mat)
```

Arguments

`var`: PSD variable of new PSD term.

`mat`: coefficient matrix of new PSD term.

PsdExpr::AddTerm()

Add a PSD term to PSD expression object.

Synopsis

```
void AddTerm(const PsdVar &var, const SymMatExpr &expr)
```

Arguments

`var`: PSD variable of new PSD term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

PsdExpr::AddTerms()

Add linear terms to PSD expression object.

Synopsis

```
int AddTerms(
    const VarArray &vars,
    double *pCoeff,
    int len)
```

Arguments

`vars`: variables for added linear terms.

`pCoeff`: coefficient array for added linear terms.

`len`: length of coefficient array.

Return

number of added linear terms.

PsdExpr::AddTerms()

Add PSD terms to PSD expression object.

Synopsis

```
int AddTerms(const PsdVarArray &vars, const SymMatrixArray &mats)
```

Arguments

`vars`: PSD variables for added PSD terms.

`mats`: coefficient matrixes for added PSD terms.

Return

number of added PSD terms.

PsdExpr::Clone()

Deep copy PSD expression object.

Synopsis

```
PsdExpr Clone()
```

Return

cloned PSD expression object.

PsdExpr::Evaluate()

evaluate PSD expression after solving

Synopsis

```
double Evaluate()
```

Return

value of PSD expression

PsdExpr::GetCoeff()

Get coefficient from the i-th term in PSD expression.

Synopsis

```
SymMatExpr &GetCoeff(int i)
```

Arguments

`i`: index of the PSD term.

Return

coefficient of the i-th PSD term in PSD expression object.

PsdExpr::GetConstant()

Get constant in PSD expression.

Synopsis

```
double GetConstant()
```

Return

constant in PSD expression.

PsdExpr::GetLinExpr()

Get linear expression in PSD expression.

Synopsis

```
Expr &GetLinExpr()
```

Return

linear expression object.

PsdExpr::GetPsdVar()

Get the PSD variable from the i-th term in PSD expression.

Synopsis

```
PsdVar &GetPsdVar(int i)
```

Arguments

`i`: index of the term.

Return

the first variable of the i-th term in PSD expression object.

PsdExpr::operator*=()

Multiply a constant to self.

Synopsis

```
void operator*=(double c)
```

Arguments

`c`: constant multiplier.

PsdExpr::operator*()

Multiply constant and return new expression.

Synopsis

```
PsdExpr operator*(double c)
```

Arguments

`c`: constant multiplier.

Return

result expression.

PsdExpr::operator+=()

Add an expression to self.

Synopsis

```
void operator+=(const PsdExpr &expr)
```

Arguments

`expr`: expression to be added.

PsdExpr::operator+()

Add expression and return new expression.

Synopsis

```
PsdExpr operator+(const PsdExpr &other)
```

Arguments

`other`: other expression to add.

Return

result expression.

PsdExpr::operator-=()

Substract an expression from self.

Synopsis

```
void operator-=(const PsdExpr &expr)
```

Arguments

`expr`: expression to be substracted.

PsdExpr::operator-()

Substract expression and return new expression.

Synopsis

```
PsdExpr operator-(const PsdExpr &other)
```

Arguments

`other`: other expression to substract.

Return

result expression.

PsdExpr::Remove()

Remove i-th term from PSD expression object.

Synopsis

```
void Remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

PsdExpr::Remove()

Remove the term associated with variable from PSD expression.

Synopsis

```
void Remove(const Var &var)
```

Arguments

`var`: a variable whose term should be removed.

PsdExpr::Remove()

Remove the term associated with PSD variable from PSD expression.

Synopsis

```
void Remove(const PsdVar &var)
```

Arguments

`var`: a PSD variable whose term should be removed.

PsdExpr::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(size_t n)
```

Arguments

`n`: minimum capacity for PSD expression object.

PsdExpr::SetCoeff()

Set coefficient matrix of the i-th term in PSD expression.

Synopsis

```
void SetCoeff(int i, const SymMatrix &mat)
```

Arguments

`i`: index of the PSD term.

`mat`: coefficient matrix of the term.

PsdExpr::SetConstant()

Set constant for the PSD expression.

Synopsis

```
void SetConstant(double constant)
```

Arguments

`constant`: the value of the constant.

PsdExpr::Size()

Get number of PSD terms in expression.

Synopsis

```
size_t Size()
```

Return

number of PSD terms.

18.4.33 PsdConstraint

COPT PSD constraint object. PSD constraints are always associated with a particular model. User creates a PSD constraint object by adding a PSD constraint to model, rather than by constructor of PsdConstraint class.

PsdConstraint::Get()

Get attribute value of the PSD constraint. Support related PSd attributes.

Synopsis

```
double Get(const char *szAttr)
```

Arguments

`szAttr`: name of queried attribute.

Return

attribute value.

PsdConstraint::GetIdx()

Get index of the PSD constraint.

Synopsis

```
int GetIdx()
```

Return

the index of the PSD constraint.

PsdConstraint::GetName()

Get name of the PSD constraint.

Synopsis

```
const char *GetName()
```

Return

the name of the PSD constraint.

PsdConstraint::Remove()

Remove this PSD constraint from model.

Synopsis

```
void Remove()
```

PsdConstraint::Set()

Set attribute value of the PSD constraint. Support related PSD attributes.

Synopsis

```
void Set(const char *szAttr, double value)
```

Arguments

`szAttr`: name of queried attribute.

`value`: new value.

PsdConstraint::SetName()

Set name of a PSD constraint.

Synopsis

```
void SetName(const char *szName)
```

Arguments

`szName`: the name to set.

18.4.34 PsdConstrArray

COPT PSD constraint array object. To store and access a set of *PsdConstraint* objects, Cardinal Optimizer provides PsdConstrArray class, which defines the following methods.

PsdConstrArray::GetPsdConstr()

Get idx-th PSD constraint object.

Synopsis

```
PsdConstraint &GetPsdConstr(int idx)
```

Arguments

`idx`: index of the PSD constraint.

Return

PSD constraint object with index idx.

PsdConstrArray::PushBack()

Add a PSD constraint object to PSD constraint array.

Synopsis

```
void PushBack(const PsdConstraint &constr)
```

Arguments

`constr`: a PSD constraint object.

PsdConstrArray::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for PSD constraint objects.

PsdConstrArray::Size()

Get the number of PSD constraint objects.

Synopsis

```
int Size()
```

Return

number of PSD constraint objects.

18.4.35 PsdConstrBuilder

COPT PSD constraint builder object. To help building a PSD constraint, given a PSD expression, constraint sense and right-hand side value, Cardinal Optimizer provides PsdConstrBuilder class, which defines the following methods.

PsdConstrBuilder::GetPsdExpr()

Get expression associated with PSD constraint.

Synopsis

```
const PsdExpr &GetPsdExpr()
```

Return

PSD expression object.

PsdConstrBuilder::GetRange()

Get range from lower bound to upper bound of range constraint.

Synopsis

```
double GetRange()
```

Return

length from lower bound to upper bound of the constraint.

PsdConstrBuilder::GetSense()

Get sense associated with PSD constraint.

Synopsis

```
char GetSense()
```

Return

PSD constraint sense.

PsdConstrBuilder::Set()

Set detail of a PSD constraint to its builder object.

Synopsis

```
void Set(
    const PsdExpr &expr,
    char sense,
    double rhs)
```

Arguments

`expr`: expression object at one side of the PSD constraint.

`sense`: PSD constraint sense, other than COPT_RANGE.

`rhs`: constant at right side of the PSD constraint.

PsdConstrBuilder::SetRange()

Set a range constraint to its builder.

Synopsis

```
void SetRange(const PsdExpr &expr, double range)
```

Arguments

`expr`: PSD expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must greater than 0.

18.4.36 PsdConstrBuilderArray

COPT PSD constraint builder array object. To store and access a set of *PsdConstrBuilder* objects, Cardinal Optimizer provides PsdConstrBuilderArray class, which defines the following methods.

PsdConstrBuilderArray::GetBuilder()

Get idx-th PSD constraint builder object.

Synopsis

```
PsdConstrBuilder &GetBuilder(int idx)
```

Arguments

`idx`: index of the PSD constraint builder.

Return

PSD constraint builder object with index idx.

PsdConstrBuilderArray::PushBack()

Add a PSD constraint builder object to PSD constraint builder array.

Synopsis

```
void PushBack(const PsdConstrBuilder &builder)
```

Arguments

`builder`: a PSD constraint builder object.

PsdConstrBuilderArray::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for PSD constraint builder object.

PsdConstrBuilderArray::Size()

Get the number of PSD constraint builder objects.

Synopsis

```
int Size()
```

Return

number of PSD constraint builder objects.

18.4.37 SymMatrix

COPT symmetric matrix object. Symmetric matrices are always associated with a particular model. User creates a symmetric matrix object by adding a symmetric matrix to model, rather than by constructor of SymMatrix class.

Symmetric matrices are used as coefficient matrices of PSD terms in PSD expressions, PSD constraints or PSD objectives.

SymMatrix::GetDim()

Get the dimension of a symmetric matrix.

Synopsis

```
int GetDim()
```

Return

dimension of a symmetric matrix.

SymMatrix::GetIdx()

Get the index of a symmetric matrix.

Synopsis

```
int GetIdx()
```

Return

index of a symmetric matrix.

18.4.38 SymMatrixArray

COPT symmetric matrix object. To store and access a set of *SymMatrix* objects, Cardinal Optimizer provides SymMatrixArray class, which defines the following methods.

SymMatrixArray::GetMatrix()

Get i-th SymMatrix object.

Synopsis

```
SymMatrix &GetMatrix(int idx)
```

Arguments

`idx`: index of the SymMatrix object.

Return

SymMatrix object with index idx.

SymMatrixArray::PushBack()

Add a SymMatrix object to SymMatrix array.

Synopsis

```
void PushBack(const SymMatrix &mat)
```

Arguments

`mat`: a SymMatrix object.

SymMatrixArray::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for symmetric matrix object.

SymMatrixArray::Size()

Get the number of SymMatrix objects.

Synopsis

```
int Size()
```

Return

number of SymMatrix objects.

18.4.39 SymMatExpr

COPT symmetric matrix expression object. A symmetric matrix expression is a linear combination of symmetric matrices, which is still a symmetric matrix. However, by doing so, we are able to delay computing the final matrix until setting PSD constraints or PSD objective.

SymMatExpr::SymMatExpr()

Constructor of a symmetric matrix expression.

Synopsis

```
SymMatExpr()
```

SymMatExpr::SymMatExpr()

Constructor of a symmetric matrix expression with one term.

Synopsis

```
SymMatExpr(const SymMatrix &mat, double coeff)
```

Arguments

`mat`: symmetric matrix of the added term.

`coeff`: optional, coefficent for the added term. Its default value is 1.0.

SymMatExpr::AddSymMatExpr()

Add a symmetric matrix expression to self.

Synopsis

```
void AddSymMatExpr(const SymMatExpr &expr, double mult)
```

Arguments

`expr`: symmetric matrix expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

SymMatExpr::AddTerm()

Add a term to symmetric matrix expression object.

Synopsis

```
bool AddTerm(const SymMatrix &mat, double coeff)
```

Arguments

`mat`: symmetric matrix of the new term.

`coeff`: coefficient of the new term.

Return

True if the term is added successfully.

SymMatExpr::AddTerms()

Add multiple terms to expression object.

Synopsis

```
int AddTerms(
    const SymMatrixArray &mats,
    double *pCoeff,
    int len)
```

Arguments

`mats`: symmetric matrix array object for added terms.

`pCoeff`: coefficient array for added terms.

`len`: length of coefficient array.

Return

Number of added terms. If negative, fail to add one of terms.

SymMatExpr::Clone()

Deep copy symmetric matrix expression object.

Synopsis

```
SymMatExpr Clone()
```

Return

cloned expression object.

SymMatExpr::GetCoeff()

Get coefficient of the i-th term in expression object.

Synopsis

```
double GetCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term.

SymMatExpr::GetDim()

Get dimension of symmetric matrix in expression.

Synopsis

```
int GetDim()
```

Return

dimension of symmetric matrix.

SymMatExpr::GetSymMat()

Get symmetric matrix of the i-th term in expression object.

Synopsis

```
SymMatrix &GetSymMat(int i)
```

Arguments

`i`: index of the term.

Return

the symmetric matrix of the i-th term.

SymMatExpr::operator*=()

Multiply a constant to self.

Synopsis

```
void operator*=(double c)
```

Arguments

`c`: constant multiplier.

SymMatExpr::operator*()

Multiply constant and return new expression.

Synopsis

```
SymMatExpr operator*(double c)
```

Arguments

`c`: constant multiplier.

Return

result expression.

SymMatExpr::operator+=()

Add a symmetric matrix expression to self.

Synopsis

```
void operator+=(const SymMatExpr &expr)
```

Arguments

`expr`: symmetric matrix expression to be added.

SymMatExpr::operator+()

Add expression and return new expression.

Synopsis

```
SymMatExpr operator+(const SymMatExpr &other)
```

Arguments

`other`: other expression to add.

Return

result expression.

SymMatExpr::operator-=()

Substract a symmetric matrix expression from self.

Synopsis

```
void operator-=(const SymMatExpr &expr)
```

Arguments

`expr`: symmetric matrix to be substracted.

SymMatExpr::operator-()

Substract expression and return new expression.

Synopsis

```
SymMatExpr operator-(const SymMatExpr &other)
```

Arguments

`other`: other expression to substract.

Return

result expression.

SymMatExpr::Remove()

Remove i-th term from expression object.

Synopsis

```
void Remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

SymMatExpr::Remove()

Remove the term associated with the symmetric matrix.

Synopsis

```
void Remove(const SymMatrix &mat)
```

Arguments

`mat`: a symmetric matrix whose term should be removed.

SymMatExpr::Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(size_t n)
```

Arguments

`n`: minimum capacity for expression object.

SymMatExpr::SetCoeff()

Set coefficient for the i-th term in expression object.

Synopsis

```
void SetCoeff(int i, double val)
```

Arguments

`i`: index of the term.

`val`: coefficient of the term.

SymMatExpr::Size()

Get number of terms in expression.

Synopsis

```
size_t Size()
```

Return

number of terms.

18.4.40 CallbackBase

COPT Callback abstract base object. Users must implment its virtual method `virtual void CallbackBase::callback()` to instantiate an instance, which pass to `Model::SetCallback(ICallback* pcb, int cbctx)` as the first parameter. Subclass of CallbackBase inherits the following member methods:

CallbackBase::AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(
    const Expr &lhs,
    char sense,
    double rhs)
```

Arguments

`lhs`: expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

CallbackBase::AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(
    const Expr &lhs,
    char sense,
    const Expr &rhs)
```

Arguments

`lhs`: left hand side expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy contraint.

CallbackBase::AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(const ConstrBuilder &builder)
```

Arguments

`builder`: builder for lazy contraint.

CallbackBase::AddLazyConstrs()

Add lazy constraints to model.

Synopsis

```
void AddLazyConstrs(const ConstrBuilderArray &builders)
```

Arguments

`builders`: array of builders for lazy contraints.

CallbackBase::AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(
    const Expr &lhs,
    char sense,
    double rhs)
```

Arguments

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

CallbackBase::AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(
    const Expr &lhs,
    char sense,
    const Expr &rhs)
```

Arguments

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

CallbackBase::AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(const ConstrBuilder &builder)
```

Arguments

`builder`: builder for user cut.

CallbackBase::AddUserCuts()

Add user cuts to model.

Synopsis

```
void AddUserCuts(const ConstrBuilderArray &builders)
```

Arguments

`builders`: array of builders for user cuts.

CallbackBase::GetDblInfo()

Get double value of given information name in callback.

Synopsis

```
double GetDblInfo(const char *cbinfo)
```

Arguments

`cbinfo`: name of callback info.

Return

value of desired information.

CallbackBase::GetIncumbent()

Get best feasible solution of given variable in callback.

Synopsis

```
double GetIncumbent(Var &var)
```

Arguments

`var`: given variable.

Return

best feasible solution of given variable.

CallbackBase::GetIncumbent()

Get best feasible solution of variables in callback.

Synopsis

```
int GetIncumbent(VarArray &vars, double *pOut)
```

Arguments

`vars`: an array of variables.

`pOut`: best feasible solution of desired variables.

Return

the number of valid variables. If failed, return -1.

CallbackBase::GetIncumbent()

Get best feasible solution of all variables in callback.

Synopsis

```
int GetIncumbent(double *pOut, int len)
```

Arguments

`pOut`: optional, output best feasible solution of all variables.

`len`: the length of output array. The solution is written up to number of len.

Return

number of all variables. Return -1 if error occurs.

CallbackBase::GetIntInfo()

Get integer value of given information name in callback.

Synopsis

```
int GetIntInfo(const char *cbinfo)
```

Arguments

`cbinfo`: name of callback info.

Return

value of desired information.

CallbackBase::GetRelaxSol()

Get LP-relaxation solution of given variable in callback.

Synopsis

```
double GetRelaxSol(Var &var)
```

Arguments

`var`: given variable.

Return

LP-relaxation solution of given variable.

CallbackBase::GetRelaxSol()

Get LP-relaxation solution of variables in callback.

Synopsis

```
int GetRelaxSol(VarArray &vars, double *pOut)
```

Arguments

`vars`: an array of variables.

`pOut`: LP-relaxation solution of desired variables.

Return

the number of valid variables. If failed, return -1.

CallbackBase::GetRelaxSol()

Get LP-relaxation solution of all variables in callback.

Synopsis

```
int GetRelaxSol(double *pOut, int len)
```

Arguments

`pOut`: optional, output LP-relaxation solution of all variables.

`len`: the length of output array. The solution is written up to number of len.

Return

number of all variables. Return -1 if error occurs.

CallbackBase::GetSolution()

Get solution of given variable in callback.

Synopsis

```
double GetSolution(Var &var)
```

Arguments

`var`: given variable.

Return

solution of desired variable.

CallbackBase::GetSolution()

Get solution of variables in callback.

Synopsis

```
int GetSolution(VarArray &vars, double *pOut)
```

Arguments

`vars`: an array of variables.

`pOut`: solution of desired variables.

Return

the number of valid variables. If failed, return -1.

CallbackBase::GetSolution()

Get solution of all variables in callback.

Synopsis

```
int GetSolution(double *pOut, int len)
```

Arguments

`pOut`: optional, output solution of all variables.

`len`: the length of output array. The solution is written up to number of len.

Return

number of all variables. Return -1 if error occurs.

CallbackBase::Interrupt()

Interrupt solving problems in callback

Synopsis

```
void Interrupt()
```

CallbackBase::LoadSolution()

Load customized solution to model.

Synopsis

```
double LoadSolution()
```

Return

objective value of given solution.

CallbackBase::SetSolution()

Set solution of a given variable in callback.

Synopsis

```
void SetSolution(Var &var, double val)
```

Arguments

`var`: a variable object.

`val`: double value.

CallbackBase::SetSolution()

Set solution of variables in callback.

Synopsis

```
void SetSolution(
    VarArray &vars,
    const double *vals,
    int len)
```

Arguments

`vars`: an array of variable objects.

`vals`: an array of double values.

`len`: length of array of double values.

CallbackBase::Where()

Get context in callback.

Synopsis

```
int Where()
```

Return

integer value of context.

18.4.41 ProbBuffer

Buffer object for COPT problem. ProbBuffer object holds the (MPS) problem in string format.

ProbBuffer::ProbBuffer()

Constructor of ProbBuffer object.

Synopsis

```
ProbBuffer(int sz)
```

Arguments

`sz`: initial size of the problem buffer.

ProbBuffer::GetData()

Get string of problem in problem buffer.

Synopsis

```
char *GetData()
```

Return

string of problem in problem buffer.

ProbBuffer::Resize()

Resize buffer to given size, and zero-ended

Synopsis

```
void Resize(int sz)
```

Arguments

`sz`: new buffer size.

ProbBuffer::Size()

Get the size of problem buffer.

Synopsis

```
int Size()
```

Return

size of problem buffer.

Chapter 19

C# API Reference

The **Cardinal Optimizer** provides C# API library. This chapter documents all COPT C# constants and API functions for C# applications.

19.1 Constants

There are four types of constants defined in **Cardinal Optimizer**. They are general constants, information constants, attributes and parameters.

19.1.1 General Constants

For the contents of C# general constants, see *General Constants*.

General constants are defined in `Consts` class. User may refer general constants with namespace, that is, `Copt.Consts.XXXX`.

19.1.2 Attributes

For the contents of C# attribute constants, see *Attributes*.

All COPT C# attributes are defined in `DblAttr` and `IntAttr` classes. User may refer double attributes by `Copt.DblAttr.XXXX`, and integer attributes by `Copt.IntAttr.XXXX`.

In the C# API, user can get the attribute value by specifying the attribute name. The two functions of obtaining attribute values are as follows, please refer to *C# API: Model Class* for details.

- `Model.GetIntAttr()`: Get value of a COPT integer attribute.
- `Model.GetDblAttr()`: Get value of a COPT double attribute.

19.1.3 Information

For the content of C# information constants, see *Information*.

In the C# API, information constants are defined in the `DblInfo` class. Users can access information constants through the prefix `Copt` in the namespace (usually can be omitted) `Copt.DblInfo.`

For instance, `Copt.DblInfo.Obj` is the coefficients of variables in the objective function.

19.1.4 Callback Information

For the content of C# API callback information class constants, see *Callback Information*.

In the C# API, callback-related information constants are defined in the `CbInfo` class. Users can access information constants through the prefix `Copt` in the namespace (usually can be omitted) `Copt.CbInfo.`

For instance, `Copt.CbInfo.BestObj` is the current best objective.

19.1.5 Parameters

For the contents of C# parameters constants, see *Parameters*.

All COPT C# parameters are defined in DblParam and IntParam classes. User may refer double parameters by `Copt.DblParam.XXXX`, and integer parameters by `Copt.IntParam.XXXX`.

In the C# API, user can get and set the parameter value by specifying the parameter name. The provided functions are as follows, please refer to *C# API: Model Class* for details.

- Get detailed information of the specified parameter (current value/max/min): `Model.GetParamInfo()`
- Get the current value of the specified integer/double parameter: `Model.GetIntParam()` / `Model.GetDblParam()`
- Set the specified integer/double parameter value: `Model.SetIntParam()` / `Model.SetDblParam()`

19.2 C# Modeling Classes

This chapter documents COPT C# interface. Users may refer to C# classes described below for details of how to construct and solve C# models.

19.2.1 Envr

Essentially, any C# application using Cardinal Optimizer should start with a COPT environment. COPT models are always associated with a COPT environment. User must create an environment object before populating models. User generally only need a single environment object in program.

Envr.Envr()

Constructor of COPT Envr object.

Synopsis

```
Envr()
```

Envr.Envr()

Constructor of COPT Envr object, given a license folder.

Synopsis

```
Envr(string licDir)
```

Arguments

`licDir`: directory having local license or client config file.

Envr.Envr()

Constructor of COPT Envr object, given an Envr config object.

Synopsis

```
Envr(EnvrConfig config)
```

Arguments

`config`: Envr config object holding settings for remote connection.

Envr.Close()

close remote connection and token becomes invalid for all problems in current envr.

Synopsis

```
void Close()
```

Envr.CreateModel()

Create a model object.

Synopsis

```
Model CreateModel(string name)
```

Arguments

`name`: customized model name.

Return

a model object.

19.2.2 EnvrConfig

If user connects to COPT remote services, such as floating token server or compute cluster, it is necessary to add config settings with EnvrConfig object.

EnvrConfig.EnvrConfig()

Constructor of envr config object.

Synopsis

```
EnvrConfig()
```

EnvrConfig.Set()

Set config settings in terms of name-value pair.

Synopsis

```
void Set(string name, string value)
```

Arguments

`name`: keyword of a config setting.

`value`: value of a config setting.

19.2.3 Model

In general, a COPT model consists of a set of variables, a (linear) objective function on these variables, a set of constraints on there varaibles, etc. COPT model class encapsulates all required methods for constructing a COPT model.

Model.Model()

Constructor of model.

Synopsis

```
Model(Envr env, string name)
```

Arguments

`env`: associated environment object.

`name`: string of model name.

Model.AddCone()

Add a cone constraint to model.

Synopsis

```
Cone AddCone(
    int dim,
    int type,
    char[] pvtype,
    string prefix)
```

Arguments

`dim`: dimension of the cone constraint.

`type`: type of the cone constraint.

`pvtype`: type of variables in the cone.

`prefix`: optional, name prefix of variables in the cone, default value is "ConeV".

Return

new cone constraint object.

Model.AddCone()

Add a cone constraint to model.

Synopsis

```
Cone AddCone(ConeBuilder builder)
```

Arguments

`builder`: builder for new cone constraint.

Return

new cone constraint object.

Model.AddCone()

Add a cone constraint to model.

Synopsis

```
Cone AddCone(Var[] vars, int type)
```

Arguments

`vars`: variables that participate in the cone constraint.

`type`: type of the cone constraint.

Return

new cone constraint object.

Model.AddCone()

Add a cone constraint to model.

Synopsis

```
Cone AddCone(VarArray vars, int type)
```

Arguments

`vars`: variables that participate in the cone constraint.

`type`: type of a cone constraint.

Return

new cone constraint object.

Model.AddConstr()

Add a linear constraint to model.

Synopsis

```
Constraint AddConstr(
    Expr expr,
    char sense,
    double rhs,
    string name)
```

Arguments

`expr`: expression for the new contraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side value for the new constraint.

`name`: optional, name of new constraint.

Return

new constraint object.

Model.AddConstr()

Add a linear constraint to model.

Synopsis

```
Constraint AddConstr(
    Expr expr,
    char sense,
    Var var,
    string name)
```

Arguments

`expr`: expression for the new contraint.

`sense`: sense for new linear constraint, other than range sense.

`var`: variable as right hand side for the new constraint.

`name`: optional, name of new constraint.

Return

new constraint object.

Model.AddConstr()

Add a linear constraint to model.

Synopsis

```
Constraint AddConstr(
    Expr lhs,
    char sense,
    Expr rhs,
    string name)
```

Arguments

`lhs`: left hand side expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side expression for the new constraint.

`name`: optional, name of new constraint.

Return

new constraint object.

Model.AddConstr()

Add a linear constraint to model.

Synopsis

```
Constraint AddConstr(
    Expr expr,
    double lb,
    double ub,
    string name)
```

Arguments

`expr`: expression for the new constraint.

`lb`: lower bound for the new constraint.

`ub`: upper bound for the new constraint

`name`: optional, name of new constraint.

Return

new constraint object.

Model.AddConstr()

Add a linear constraint to a model.

Synopsis

```
Constraint AddConstr(ConstrBuilder builder, string name)
```

Arguments

`builder`: builder for the new constraint.

`name`: optional, name of new constraint.

Return

new constraint object.

Model.AddConstrs()

Add linear constraints to model.

Synopsis

```
ConstrArray AddConstrs(
    int count,
    char[] senses,
    double[] rhss,
    string prefix)
```

Arguments

`count`: number of constraints added to model.

`senses`: sense array for new linear constraints, other than range sense.

`rhss`: right hand side values for new variables.

`prefix`: optional, name prefix for new constraints, default value is 'R'.

Return

array of new constraint objects.

Model.AddConstrs()

Add linear constraints to a model.

Synopsis

```
ConstrArray AddConstrs(
    int count,
    double[] lbs,
    double[] ubs,
    string prefix)
```

Arguments

`count`: number of constraints added to the model.

`lbs`: lower bounds of new constraints.

`ubs`: upper bounds of new constraints.

`prefix`: optional, name prefix for new constraints, default value is 'R'.

Return

array of new constraint objects.

Model.AddConstrs()

Add linear constraints to a model.

Synopsis

```
ConstrArray AddConstrs(ConstrBuilderArray builders, string prefix)
```

Arguments

`builders`: builders for new constraints.

`prefix`: optional, name prefix for new constraints, default value is 'R'.

Return

array of new constraint objects.

Model.AddDenseMat()

Add a dense symmetric matrix to a model.

Synopsis

```
SymMatrix AddDenseMat(int dim, double[] vals)
```

Arguments

`dim`: dimension of the dense symmetric matrix.

`vals`: array of non-zero elements, filled in column-wise up to len or max length of symmetric matrix.

Return

new symmetric matrix object.

Model.AddDenseMat()

Add a dense symmetric matrix to a model.

Synopsis

```
SymMatrix AddDenseMat(int dim, double val)
```

Arguments

`dim`: dimension of dense symmetric matrix.

`val`: value to fill dense symmetric matrix.

Return

new symmetric matrix object.

Model.AddDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix AddDiagMat(int dim, double val)
```

Arguments

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

Return

new diagonal matrix object.

Model.AddDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix AddDiagMat(int dim, double[] vals)
```

Arguments

`dim`: dimension of diagonal matrix.

`vals`: array of values of diagonal elements.

Return

new diagonal matrix object.

Model.AddDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix AddDiagMat(
    int dim,
    double val,
    int offset)
```

Arguments

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

`offset`: shift distance against diagonal line.

Return

new diagonal matrix object.

Model.AddDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix AddDiagMat(
    int dim,
    double[] vals,
    int offset)
```

Arguments

`dim`: dimension of diagonal matrix.

`vals`: array of values of diagonal elements.

`offset`: shift distance against diagonal line.

Return

new diagonal matrix object.

Model.AddEyeMat()

Add an identity matrix to a model.

Synopsis

```
SymMatrix AddEyeMat(int dim)
```

Arguments

`dim`: dimension of identity matrix.

Return

new identity matrix object.

Model.AddGenConstrIndicator()

Add a general constraint of type indicator to model.

Synopsis

```
GenConstr AddGenConstrIndicator(GenConstrBuilder builder)
```

Arguments

`builder`: builder for the general constraint.

Return

new general constraint object of type indicator.

Model.AddGenConstrIndicator()

Add a general constraint of type indicator to model.

Synopsis

```
GenConstr AddGenConstrIndicator(
    Var binvar,
    int binval,
    ConstrBuilder builder)
```

Arguments

`binvar`: binary indicator variable.

`binval`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`builder`: builder for linear constraint.

Return

new general constraint object of type indicator.

Model.AddGenConstrIndicator()

Add a general constraint of type indicator to model.

Synopsis

```
GenConstr AddGenConstrIndicator(
    Var binvar,
    int binval,
    Expr expr,
    char sense,
    double rhs)
```

Arguments

`binvar`: binary indicator variable.

`binval`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`expr`: expression for new linear contraint.

`sense`: sense for new linear constraint.

`rhs`: right hand side value for new linear constraint.

Return

new general constraint object of type indicator.

Model.AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(
    Expr lhs,
    char sense,
    double rhs,
    string name)
```

Arguments

`lhs`: expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

`name`: optional, name of lazy constraint.

Model.AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(
    Expr lhs,
    char sense,
    Expr rhs,
    string name)
```

Arguments

`lhs`: left hand side expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy contraint.

`name`: optional, name of lazy constraint.

Model.AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(ConstrBuilder builder, string name)
```

Arguments

`builder`: builder for lazy contraint.

`name`: optional, name of lazy constraint.

Model.AddLazyConstrs()

Add lazy constraints to model.

Synopsis

```
void AddLazyConstrs(ConstrBuilderArray builders, string prefix)
```

Arguments

`builders`: array of builders for lazy contraints.

`prefix`: name prefix of new lazy constraints.

Model.AddOnesMat()

Add a dense symmetric matrix of value one to a model.

Synopsis

```
SymMatrix AddOnesMat(int dim)
```

Arguments

`dim`: dimension of dense symmetric matrix.

Return

new symmetric matrix object.

Model.AddPsdConstr()

Add a PSD constraint to model.

Synopsis

```
PsdConstraint AddPsdConstr(
    PsdExpr expr,
    char sense,
    double rhs,
    string name)
```

Arguments

`expr`: PSD expression for new PSD contraint.

`sense`: sense for new PSD constraint.

`rhs`: double value at right side of the new PSD constraint.

`name`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model.AddPsdConstr()

Add a PSD constraint to model.

Synopsis

```
PsdConstraint AddPsdConstr(
    PsdExpr expr,
    double lb,
    double ub,
    string name)
```

Arguments

`expr`: expression for new PSD constraint.

`lb`: lower bound for ew PSD constraint.

`ub`: upper bound for new PSD constraint

`name`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model.AddPsdConstr()

Add a PSD constraint to model.

Synopsis

```
PsdConstraint AddPsdConstr(
    PsdExpr lhs,
    char sense,
    PsdExpr rhs,
    string name)
```

Arguments

`lhs`: PSD expression at left side of new PSD constraint.

`sense`: sense for new PSD constraint.

`rhs`: PSD expression at right side of new PSD constraint.

`name`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model.AddPsdConstr()

Add a PSD constraint to a model.

Synopsis

```
PsdConstraint AddPsdConstr(PsdConstrBuilder builder, string name)
```

Arguments

`builder`: builder for new PSD constraint.

`name`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model.AddPsdVar()

Add a new PSD variable to model.

Synopsis

```
PsdVar AddPsdVar(int dim, string name)
```

Arguments

`dim`: dimension of new PSD variable.

`name`: name of new PSD variable.

Return

PSD variable object.

Model.AddPsdVars()

Add new PSD variables to model.

Synopsis

```
PsdVarArray AddPsdVars(
    int count,
    int[] dims,
    string prefix)
```

Arguments

`count`: number of new PSD variables.

`dims`: array of dimensions of new PSD variables.

`prefix`: name prefix of new PSD variables, default prefix is PSD_V.

Return

array of PSD variable objects.

Model.AddQConstr()

Add a quadratic constraint to model.

Synopsis

```
QConstraint AddQConstr(
    QuadExpr expr,
    char sense,
    double rhs,
    string name)
```

Arguments

`expr`: quadratic expression for the new contraint.

`sense`: sense for new quadratic constraint.

`rhs`: double value at right side of the new quadratic constraint.

`name`: optional, name of new quadratic constraint.

Return

new quadratic constraint object.

Model.AddQConstr()

Add a quadratic constraint to model.

Synopsis

```
QConstraint AddQConstr(
    QuadExpr lhs,
    char sense,
    QuadExpr rhs,
    string name)
```

Arguments

`lhs`: quadratic expression at left side of the new quadratic constraint.

`sense`: sense for new quadratic constraint.

`rhs`: quadratic expression at right side of the new quadratic constraint.

`name`: optional, name of new quadratic constraint.

Return

new quadratic constraint object.

Model.AddQConstr()

Add a quadratic constraint to a model.

Synopsis

```
QConstraint AddQConstr(QConstrBuilder builder, string name)
```

Arguments

`builder`: builder for the new quadratic constraint.

`name`: optional, name of new quadratic constraint.

Return

new quadratic constraint object.

Model.AddSos()

Add a SOS constraint to model.

Synopsis

```
Sos AddSos(
    Var[] vars,
    double[] weights,
    int type)
```

Arguments

`vars`: variables that participate in the SOS constraint.

`weights`: weights for variables in the SOS constraint.

`type`: type of SOS constraint.

Return

new SOS constraint object.

Model.AddSos()

Add a SOS constraint to model.

Synopsis

```
Sos AddSos(
    VarArray vars,
    double[] weights,
    int type)
```

Arguments

`vars`: variables that participate in the SOS constraint.

`weights`: weights for variables in the SOS constraint.

`type`: type of SOS constraint.

Return

new SOS constraint object.

Model.AddSos()

Add a SOS constraint to model.

Synopsis

```
Sos AddSos(SosBuilder builder)
```

Arguments

`builder`: builder for new SOS constraint.

Return

new SOS constraint object.

Model.AddSparseMat()

Add a sparse symmetric matrix to a model.

Synopsis

```
SymMatrix AddSparseMat(
    int dim,
    int nElems,
    int[] rows,
    int[] cols,
    double[] vals)
```

Arguments

`dim`: dimension of the sparse symmetric matrix.

`nElems`: number of non-zero elements in the sparse symmetric matrix.

`rows`: array of row indexes of non-zero elements.

`cols`: array of col indexes of non-zero elements.

`vals`: array of values of non-zero elements.

Return

new symmetric matrix object.

Model.AddSymMat()

Given a symmetric matrix expression, add results matrix to model.

Synopsis

```
SymMatrix AddSymMat(SymMatExpr expr)
```

Arguments

`expr`: symmetric matrix expression object.

Return

results symmetric matrix object.

Model.AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(
    Expr lhs,
    char sense,
    double rhs,
    string name)
```

Arguments

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

`name`: optional, name of user cut.

Model.AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(
    Expr lhs,
    char sense,
    Expr rhs,
    string name)
```

Arguments

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

`name`: optional, name of user cut.

Model.AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(ConstrBuilder builder, string name)
```

Arguments

`builder`: builder for user cut.

`name`: optional, name of user cut.

Model.AddUserCuts()

Add user cuts to model.

Synopsis

```
void AddUserCuts(ConstrBuilderArray builders, string prefix)
```

Arguments

`builders`: array of builders for user cuts.

`prefix`: name prefix of new user cuts.

Model.AddVar()

Add a variable and the associated non-zero coefficients as column.

Synopsis

```
Var AddVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    string name)
```

Arguments

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`name`: optional, name for new variable.

Return

new variable object.

Model.AddVar()

Add a variable and the associated non-zero coefficients as column.

Synopsis

```
Var AddVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    Column col,
    string name)
```

Arguments

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`col`: column object for specifying a set of constraints to which the variable belongs.

`name`: optional, name for new variable.

Return

new variable object.

Model.AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    int count,
    char vtype,
    string prefix)
```

Arguments

`count`: the number of variables to add.

`vtype`: variable types for new variables.

`prefix`: optional, prefix part for names of new variables, default value is 'C'.

Return

array of new variable objects.

Model.AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    int count,
    double lb,
    double ub,
    double obj,
    char vtype,
    string prefix)
```

Arguments

`count`: the number of variables to add.

`lb`: lower bound for new variables.

`ub`: upper bound for new variables.

`obj`: objective coefficient for new variables.

`vtype`: variable type for new variables.

`prefix`: optional, prefix part for names of new variables, default value is 'C'.

Return

array of new variable objects.

Model.AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    int count,
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    string prefix)
```

Arguments

`count`: the number of variables to add.

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`prefix`: optional, prefix part for names of new variables, default value is 'C'.

Return

array of new variable objects.

Model.AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    Column[] cols,
    string prefix)
```

Arguments

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`cols`: column objects for specifying a set of constraints to which each new variable belongs.

`prefix`: optional, prefix part for names of new variables, default value is 'C'.

Return

array of new variable objects.

Model.AddVars()

Add new variables to model.

Synopsis

```
VarArray AddVars(
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    ColumnArray cols,
    string prefix)
```

Arguments

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`cols`: columnarray for specifying a set of constraints to which each new variable belongs.

`prefix`: optional, prefix part for names of new variables, default value is 'C'.

Return

array of new variable objects.

Model.Clear()

Clear all settings including problem itself.

Synopsis

```
void Clear()
```

Model.Clone()

Deep copy a new model object.

Synopsis

```
Model Clone()
```

Return

cloned model object.

Model.ComputeIIS()

Compute IIS for model

Synopsis

```
void ComputeIIS()
```

Model.DelPsdObj()

delete PSD part of objective in model.

Synopsis

```
void DelPsdObj()
```

Model.DelQuadObj()

delete quadratic part of objective in model.

Synopsis

```
void DelQuadObj()
```

Model.FeasRelax()

Compute feasibility relaxation for infeasible model.

Synopsis

```
void FeasRelax(
    VarArray vars,
    double[] colLowPen,
    double[] colUppPen,
    ConstrArray cons,
    double[] rowBndPen,
    double[] rowUppPen)
```

Arguments

`vars`: an array of variables.

`colLowPen`: penalties for lower bounds of variables.

`colUppPen`: penalties for upper bounds of variables.

`cons`: an array of constraints.

`rowBndPen`: penalties for right hand sides of constraints.

`rowUppPen`: penalties for upper bounds of range constraints.

Model.FeasRelax()

Compute feasibility relaxation for infeasible model.

Synopsis

```
void FeasRelax(int ifRelaxVars, int ifRelaxCons)
```

Arguments

`ifRelaxVars`: whether to relax variables.

`ifRelaxCons`: whether to relax constraints.

Model.Get()

Query values of double parameter or double attribute, associated with variables.

Synopsis

```
double[] Get(string name, Var[] vars)
```

Arguments

`name`: name of double parameter or double attribute.

`vars`: a list of interested variables.

Return

values of parameter or attrribute.

Model.Get()

Query values of double parameter or double attribute, associated with variables.

Synopsis

```
double[] Get(string name, VarArray vars)
```

Arguments

`name`: name of double parameter or double attribute.

`vars`: array of interested variables.

Return

values of parameter or attrribute.

Model.Get()

Query values of parameter or attribute, associated with constraints.

Synopsis

```
double[] Get(string name, Constraint[] constrs)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: a list of interested constraints.

Return

values of parameter or attrribute.

Model.Get()

Query values of parameter or attribute, associated with constraints.

Synopsis

```
double[] Get(string name, ConstrArray constrs)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: array of interested constraints.

Return

values of parameter or attrribute.

Model.Get()

Query values of parameter or attribute, associated with quadratic constraints.

Synopsis

```
double[] Get(string name, QConstraint[] constrs)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: a list of interested quadratic constraints.

Return

values of parameter or attrribute.

Model.Get()

Query values of parameter or attribute, associated with quadratic constraints.

Synopsis

```
double[] Get(string name, QConstrArray constrs)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: array of interested quadratic constraints.

Return

values of parameter or attrribute.

Model.Get()

Query values of parameter or attribute, associated with PSD constraints.

Synopsis

```
double[] Get(string name, PsdConstraint[] constrs)
```

Arguments

`name`: name of double parameter or attribute.

`constrs`: a list of desired PSD constraints.

Return

output array of parameter or attribute values.

Model.Get()

Query values of parameter or attribute, associated with PSD constraints.

Synopsis

```
double[] Get(string name, PsdConstrArray constrs)
```

Arguments

`name`: name of double parameter or attribute.

`constrs`: a list of desired PSD constraints.

Return

output array of parameter or attribute values.

Model.GetCoeff()

Get the coefficient of variable in linear constraint.

Synopsis

```
double GetCoeff(Constraint constr, Var var)
```

Arguments

`constr`: The requested constraint.

`var`: The requested variable.

Return

The requested coefficient.

Model.GetCol()

Get a column object that have a list of constraints in which the variable participates.

Synopsis

```
Column GetCol(Var var)
```

Arguments

`var`: a variable object.

Return

a column object associated with a variable.

Model.GetColBasis()

Get status of column basis.

Synopsis

```
int[] GetColBasis()
```

Return

basis status.

Model.GetCone()

Get a cone constraint of given index in model.

Synopsis

```
Cone GetCone(int idx)
```

Arguments

`idx`: index of the desired cone constraint.

Return

the desired cone constraint object.

Model.GetConeBuilders()

Get builders of all cone constraints in model.

Synopsis

```
ConeBuilderArray GetConeBuilders()
```

Return

array object of cone constraint builders.

Model.GetConeBuilders()

Get builders of given cone constraints in model.

Synopsis

```
ConeBuilderArray GetConeBuilders(Cone[] cones)
```

Arguments

`cones`: array of cone constraints.

Return

array object of desired cone constraint builders.

Model.GetConeBuilders()

Get builders of given cone constraints in model.

Synopsis

```
ConeBuilderArray GetConeBuilders(ConeArray cones)
```

Arguments

`cones`: array of cone constraints.

Return

array object of desired cone constraint builders.

Model.GetCones()

Get all cone constraints in model.

Synopsis

```
ConeArray GetCones()
```

Return

array object of cone constraints.

Model.GetConstr()

Get a constraint of given index in model.

Synopsis

```
Constraint GetConstr(int idx)
```

Arguments

`idx`: index of the desired constraint.

Return

the desired constraint object.

Model.GetConstrBuilder()

Get builder of a constraint in model, including variables and associated coefficients, sense and RHS.

Synopsis

```
ConstrBuilder GetConstrBuilder(Constraint constr)
```

Arguments

`constr`: a constraint object.

Return

constraint builder object.

Model.GetConstrBuilders()

Get builders of all constraints in model.

Synopsis

```
ConstrBuilderArray GetConstrBuilders()
```

Return

array object of constraint builders.

Model.GetConstrByName()

Get a constraint of given name in model.

Synopsis

```
Constraint GetConstrByName(string name)
```

Arguments

`name`: name of the desired constraint.

Return

the desired constraint object.

Model.GetConstrLowerIIS()

Get IIS status of lower bounds of constraints.

Synopsis

```
int[] GetConstrLowerIIS(ConstrArray constrs)
```

Arguments

`constrs`: Array of constraints.

Return

IIS status of lower bounds of constraints.

Model.GetConstrLowerIIS()

Get IIS status of lower bounds of constraints.

Synopsis

```
int[] GetConstrLowerIIS(Constraint[] constrs)
```

Arguments

`constrs`: Array of constraints.

Return

IIS status of lower bounds of constraints.

Model.GetConstrs()

Get all constraints in model.

Synopsis

```
ConstrArray GetConstrs()
```

Return

array object of constraints.

Model.GetConstrUpperIIS()

Get IIS status of upper bounds of constraints.

Synopsis

```
int[] GetConstrUpperIIS(ConstrArray constrs)
```

Arguments

`constrs`: Array of constraints.

Return

IIS status of upper bounds of constraints.

Model.GetConstrUpperIIS()

Get IIS status of upper bounds of constraints.

Synopsis

```
int[] GetConstrUpperIIS(Constraint[] constrs)
```

Arguments

`constrs`: Array of constraints.

Return

IIS status of upper bounds of constraints.

Model.GetDblAttr()

Get value of a COPT double attribute.

Synopsis

```
double GetDblAttr(string attr)
```

Arguments

`attr`: name of double attribute.

Return

value of double attribute.

Model.GetDblParam()

Get value of a COPT double parameter.

Synopsis

```
double GetDblParam(string param)
```

Arguments

`param`: name of integer parameter.

Return

value of double parameter.

Model.GetGenConstrIndicator()

Get builder of given general constraint of type indicator.

Synopsis

```
GenConstrBuilder GetGenConstrIndicator(GenConstr indicator)
```

Arguments

`indicator`: a general constraint of type indicator.

Return

builder object of general constraint of type indicator.

Model.GetIndicatorIIS()

Get IIS status of indicator constraints.

Synopsis

```
int[] GetIndicatorIIS(GenConstrArray genconstrs)
```

Arguments

`genconstrs`: Array of indicator constraints.

Return

IIS status of indicator constraints.

Model.GetIndicatorIIS()

Get IIS status of indicator constraints.

Synopsis

```
int[] GetIndicatorIIS(GenConstr[] genconstrs)
```

Arguments

`genconstrs`: Array of indicator constraints.

Return

IIS status of indicator constraints.

Model.GetIntAttr()

Get value of a COPT integer attribute.

Synopsis

```
int GetIntAttr(string attr)
```

Arguments

`attr`: name of integer attribute.

Return

value of integer attribute.

Model.GetIntParam()

Get value of a COPT integer parameter.

Synopsis

```
int GetIntParam(string param)
```

Arguments

`param`: name of integer parameter.

Return

value of integer parameter.

Model.GetLpSolution()

Get LP solution.

Synopsis

```
void GetLpSolution(
    out double[] value,
    out double[] slack,
    out double[] rowDual,
    out double[] redCost)
```

Arguments

`value`: out, solution values.

`slack`: out, slack values.

`rowDual`: out, dual values.

`redCost`: out, reduced costs.

Model.GetObjective()

Get linear expression of objective for model.

Synopsis

```
Expr GetObjective()
```

Return

an linear expression object.

Model.GetParamInfo()

Get current, default, minimum, maximum of COPT integer parameter.

Synopsis

```
void GetParamInfo(
    string name,
    out int cur,
    out int def,
    out int min,
    out int max)
```

Arguments

`name`: name of integer parameter.

`cur`: out, current value of integer parameter.

`def`: out, default value of integer parameter.

`min`: out, minimum value of integer parameter.

`max`: out, maximum value of integer parameter.

Model.GetParamInfo()

Get current, default, minimum, maximum of COPT double parameter.

Synopsis

```
void GetParamInfo(
    string name,
    out double cur,
    out double def,
    out double min,
    out double max)
```

Arguments

`name`: name of integer parameter.

`cur`: out, current value of double parameter.

`def`: out, default value of double parameter.

`min`: out, minimum value of double parameter.

`max`: out, maximum value of double parameter.

Model.GetPoolObjVal()

Get the idx-th objective value in solution pool.

Synopsis

```
double GetPoolObjVal(int idx)
```

Arguments

`idx`: Index of solution.

Return

The requested objective value.

Model.GetPoolSolution()

Get the idx-th solution in solution pool.

Synopsis

```
double[] GetPoolSolution(int idx, VarArray vars)
```

Arguments

`idx`: Index of solution.

`vars`: The requested variables.

Return

The requested solution.

Model.GetPoolSolution()

Get the idx-th solution in solution pool.

Synopsis

```
double[] GetPoolSolution(int idx, Var[] vars)
```

Arguments

`idx`: Index of solution.

`vars`: The requested variables.

Return

The requested solution.

Model.GetPsdCoeff()

Get the symmetric matrix of PSD variable in a PSD constraint.

Synopsis

```
SymMatrix GetPsdCoeff(PsdConstraint constr, PsdVar var)
```

Arguments

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

Return

The associated coefficient matrix.

Model.GetPsdConstr()

Get a quadratic constraint of given index in model.

Synopsis

```
PsdConstraint GetPsdConstr(int idx)
```

Arguments

`idx`: index of the desired quadratic constraint.

Return

the desired quadratic constraint object.

Model.GetPsdConstrBuilder()

Get builder of a PSD constraint in model, including PSD variables, sense and associated symmtric matrix.

Synopsis

```
PsdConstrBuilder GetPsdConstrBuilder(PsdConstraint constr)
```

Arguments

`constr`: a PSD constraint object.

Return

PSD constraint builder object.

Model.GetPsdConstrBuilders()

Get builders of all PSD constraints in model.

Synopsis

```
PsdConstrBuilderArray GetPsdConstrBuilders()
```

Return

array object of PSD constraint builders.

Model.GetPsdConstrByName()

Get a quadratic constraint of given name in model.

Synopsis

```
PsdConstraint GetPsdConstrByName(string name)
```

Arguments

`name`: name of the desired constraint.

Return

the desired quadratic constraint object.

Model.GetPsdConstrs()

Get all PSD constraints in model.

Synopsis

```
PsdConstrArray GetPsdConstrs()
```

Return

array object of PSD constraints.

Model.GetPsdObjective()

Get PSD objective of model.

Synopsis

```
PsdExpr GetPsdObjective()
```

Return

a PSD expression object.

Model.GetPsdRow()

Get PSD variables and associated symmetric matrix that participate in a PSD constraint.

Synopsis

```
PsdExpr GetPsdRow(PsdConstraint constr)
```

Arguments

`constr`: a PSD constraint object.

Return

PSD expression object of the PSD constraint.

Model.GetPsdVar()

Get a PSD variable of given index in model.

Synopsis

```
PsdVar GetPsdVar(int idx)
```

Arguments

`idx`: index of the desired PSD variable.

Return

the desired PSD variable object.

Model.GetPsdVarByName()

Get a PSD variable of given name in model.

Synopsis

```
PsdVar GetPsdVarByName(string name)
```

Arguments

`name`: name of the desired PSD variable.

Return

the desired PSD variable object.

Model.GetPsdVars()

Get all PSD variables in model.

Synopsis

```
PsdVarArray GetPsdVars()
```

Return

array object of PSD variables.

Model.GetQConstr()

Get a quadratic constraint of given index in model.

Synopsis

```
QConstraint GetQConstr(int idx)
```

Arguments

`idx`: index of the desired quadratic constraint.

Return

the desired quadratic constraint object.

Model.GetQConstrBuilder()

Get builder of a constraint in model, including variables and associated coefficients, sense and RHS.

Synopsis

```
QConstrBuilder GetQConstrBuilder(QConstraint constr)
```

Arguments

`constr`: a constraint object.

Return

constraint builder object.

Model.GetQConstrBuilders()

Get builders of all constraints in model.

Synopsis

```
QConstrBuilderArray GetQConstrBuilders()
```

Return

array object of constraint builders.

Model.GetQConstrByName()

Get a quadratic constraint of given name in model.

Synopsis

```
QConstraint GetQConstrByName(string name)
```

Arguments

`name`: name of the desired constraint.

Return

the desired quadratic constraint object.

Model.GetQConstrs()

Get all quadratic constraints in model.

Synopsis

```
QConstrArray GetQConstrs()
```

Return

array object of quadratic constraints.

Model.GetQuadObjective()

Get quadratic objective of model.

Synopsis

```
QuadExpr GetQuadObjective()
```

Return

a quadratic expression object.

Model.GetQuadRow()

Get two variables and associated coefficients that participate in a quadratic constraint.

Synopsis

```
QuadExpr GetQuadRow(QConstraint constr)
```

Arguments

`constr`: a quadratic constraint object.

Return

quadratic expression object of the constraint.

Model.GetRow()

Get variables that participate in a constraint, and the associated coefficients.

Synopsis

```
Expr GetRow(Constraint constr)
```

Arguments

`constr`: a constraint object.

Return

expression object of the constraint.

Model.GetRowBasis()

Get status of row basis.

Synopsis

```
int[] GetRowBasis()
```

Return

basis status.

Model.GetSolution()

Get MIP solution.

Synopsis

```
double[] GetSolution()
```

Return

solution values.

Model.GetSos()

Get a SOS constraint of given index in model.

Synopsis

```
Sos GetSos(int idx)
```

Arguments

`idx`: index of the desired SOS constraint.

Return

the desired SOS constraint object.

Model.GetSosBuilders()

Get builders of all SOS constraints in model.

Synopsis

```
SosBuilderArray GetSosBuilders()
```

Return

array object of SOS constraint builders.

Model.GetSosBuilders()

Get builders of given SOS constraints in model.

Synopsis

```
SosBuilderArray GetSosBuilders(Sos[] soss)
```

Arguments

`soss`: array of SOS constraints.

Return

array object of desired SOS constraint builders.

Model.GetSosBuilders()

Get builders of given SOS constraints in model.

Synopsis

```
SosBuilderArray GetSosBuilders(SosArray soss)
```

Arguments

`soss`: array of SOS constraints.

Return

array object of desired SOS constraint builders.

Model.GetSOSIIS()

Get IIS status of SOS constraints.

Synopsis

```
int[] GetSOSIIS(SosArray soss)
```

Arguments

`soss`: Array of SOS constraints.

Return

IIS status of SOS constraints.

Model.GetSOSIIS()

Get IIS status of SOS constraints.

Synopsis

```
int[] GetSOSIIS(Sos[] soss)
```

Arguments

`soss`: Array of SOS constraints.

Return

IIS status of SOS constraints.

Model.GetSoss()

Get all SOS constraints in model.

Synopsis

```
SosArray GetSoss()
```

Return

array object of SOS constraints.

Model.GetSymMat()

Get a symmetric matrix of given index in model.

Synopsis

```
SymMatrix GetSymMat(int idx)
```

Arguments

`idx`: index of the desired symmetric matrix.

Return

the desired symmetric matrix object.

Model.GetVar()

Get a variable of given index in model.

Synopsis

```
Var GetVar(int idx)
```

Arguments

`idx`: index of the desired variable.

Return

the desired variable object.

Model.GetVarByName()

Get a variable of given name in model.

Synopsis

```
Var GetVarByName(string name)
```

Arguments

`name`: name of the desired variable.

Return

the desired variable object.

Model.GetVarLowerIIS()

Get IIS status of lower bounds of variables.

Synopsis

```
int[] GetVarLowerIIS(VarArray vars)
```

Arguments

`vars`: Array of variables.

Return

IIS status of lower bounds of variables.

Model.GetVarLowerIIS()

Get IIS status of lower bounds of variables.

Synopsis

```
int[] GetVarLowerIIS(Var[] vars)
```

Arguments

`vars`: Array of variables.

Return

IIS status of lower bounds of variables.

Model.GetVars()

Get all variables in model.

Synopsis

```
VarArray GetVars()
```

Return

variable array object.

Model.GetVarUpperIIS()

Get IIS status of upper bounds of variables.

Synopsis

```
int[] GetVarUpperIIS(VarArray vars)
```

Arguments

`vars`: Array of variables.

Return

IIS status of upper bounds of variables.

Model.GetVarUpperIIS()

Get IIS status of upper bounds of variables.

Synopsis

```
int[] GetVarUpperIIS(Var[] vars)
```

Arguments

`vars`: Array of variables.

Return

IIS status of upper bounds of variables.

Model.Interrupt()

Interrupt optimization of current problem.

Synopsis

```
void Interrupt()
```

Model.LoadMipStart()

Load final initial values of variables to the problem.

Synopsis

```
void LoadMipStart()
```

Model.LoadTuneParam()

Load specified tuned parameters into model.

Synopsis

```
void LoadTuneParam(int idx)
```

Arguments

`idx`: Index of tuned parameters.

Model.Read()

Read problem, solution, basis, MIP start or COPT parameters from file.

Synopsis

```
void Read(string filename)
```

Arguments

`filename`: an input file name.

Model.ReadBasis()

Read basis from file.

Synopsis

```
void ReadBasis(string filename)
```

Arguments

`filename`: an input file name

Model.ReadBin()

Read problem in COPT binary format from file.

Synopsis

```
void ReadBin(string filename)
```

Arguments

`filename`: an input file name.

Model.ReadCbf()

Read problem in CBF format from file.

Synopsis

```
void ReadCbf(string filename)
```

Arguments

`filename`: an input file name.

Model.ReadLp()

Read problem in LP format from file.

Synopsis

```
void ReadLp(string filename)
```

Arguments

`filename`: an input file name.

Model.ReadMps()

Read problem in MPS format from file.

Synopsis

```
void ReadMps(string filename)
```

Arguments

`filename`: an input file name.

Model.ReadMst()

Read MIP start information from file.

Synopsis

```
void ReadMst(string filename)
```

Arguments

`filename`: an input file name.

Model.ReadParam()

Read COPT parameters from file.

Synopsis

```
void ReadParam(string filename)
```

Arguments

`filename`: an input file name.

Model.ReadSdpa()

Read problem in SDPA format from file.

Synopsis

```
void ReadSdpa(string filename)
```

Arguments

`filename`: an input file name.

Model.ReadSol()

Read solution from file.

Synopsis

```
void ReadSol(string filename)
```

Arguments

`filename`: an input file name.

Model.ReadTune()

Read tuning parameters from file.

Synopsis

```
void ReadTune(string filename)
```

Arguments

`filename`: an input file name.

Model.Remove()

Remove an array of variables from model.

Synopsis

```
void Remove(Var[] vars)
```

Arguments

`vars`: a list of variables.

Model.Remove()

Remove an array of variables from model.

Synopsis

```
void Remove(VarArray vars)
```

Arguments

`vars`: array of variables.

Model.Remove()

Remove a list of constraints from model.

Synopsis

```
void Remove(Constraint[] constrs)
```

Arguments

`constrs`: a list of constraints.

Model.Remove()

Remove a list of constraints from model.

Synopsis

```
void Remove(ConstrArray constrs)
```

Arguments

`constrs`: an array of constraints.

Model.Remove()

Remove a list of SOS constraints from model.

Synopsis

```
void Remove(Sos[] soss)
```

Arguments

`soss`: a list of SOS constraints.

Model.Remove()

Remove a list of SOS constraints from model.

Synopsis

```
void Remove(SosArray soss)
```

Arguments

`soss`: an array of SOS constraints.

Model.Remove()

Remove a list of Cone constraints from model.

Synopsis

```
void Remove(Cone[] cones)
```

Arguments

`cones`: a list of Cone constraints.

Model.Remove()

Remove a list of Cone constraints from model.

Synopsis

```
void Remove(ConeArray cones)
```

Arguments

`cones`: an array of Cone constraints.

Model.Remove()

Remove a list of gernal constraints from model.

Synopsis

```
void Remove(GenConstr[] genConstrs)
```

Arguments

`genConstrs`: a list of general constraints.

Model.Remove()

Remove a list of gernal constraints from model.

Synopsis

```
void Remove(GenConstrArray genConstrs)
```

Arguments

`genConstrs`: an array of general constraints.

Model.Remove()

Remove a list of quadratic constraints from model.

Synopsis

```
void Remove(QConstraint[] qconstrs)
```

Arguments

`qconstrs`: an array of quadratic constraints.

Model.Remove()

Remove a list of quadratic constraints from model.

Synopsis

```
void Remove(QConstrArray qconstrs)
```

Arguments

`qconstrs`: an array of quadratic constraints.

Model.Remove()

Remove a list of PSD variables from model.

Synopsis

```
void Remove(PsdVar[] vars)
```

Arguments

`vars`: an array of PSD variables.

Model.Remove()

Remove a list of PSD variables from model.

Synopsis

```
void Remove(PsdVarArray vars)
```

Arguments

`vars`: an array of PSD variables.

Model.Remove()

Remove a list of PSD constraints from model.

Synopsis

```
void Remove(PsdConstraint[] constrs)
```

Arguments

`constrs`: an array of PSD constraints.

Model.Remove()

Remove a list of PSD constraints from model.

Synopsis

```
void Remove(PsdConstrArray constrs)
```

Arguments

`constrs`: an array of PSD constraints.

Model.Reset()

Reset solution only.

Synopsis

```
void Reset()
```

Model.ResetAll()

Reset solution and additional information.

Synopsis

```
void ResetAll()
```

Model.ResetParam()

Reset parameters to default settings.

Synopsis

```
void ResetParam()
```

Model.Set()

Set values of double parameter or double attribute, associated with variables.

Synopsis

```
void Set(
    string name,
    Var[] vars,
    double[] vals)
```

Arguments

`name`: name of double parameter or double attribute.

`vars`: a list of interested variables.

`vals`: values of parameter or attrribute.

Model.Set()

Set values of double parameter or double attribute, associated with variables.

Synopsis

```
void Set(
    string name,
    VarArray vars,
    double[] vals)
```

Arguments

`name`: name of double parameter or double attribute.

`vars`: array of interested variables.

`vals`: values of parameter or attrribute.

Model.Set()

Set values of parameter or attribute, associated with constraints.

Synopsis

```
void Set(
    string name,
    Constraint[] constrs,
    double[] vals)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: a list of interested constraints.

`vals`: values of parameter or attrribute.

Model.Set()

Set values of parameter or attribute, associated with constraints.

Synopsis

```
void Set(
    string name,
    ConstrArray constrs,
    double[] vals)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: array of interested constraints.

`vals`: values of parameter or attrribute.

Model.Set()

Set values of parameter, associated with PSD constraints.

Synopsis

```
void Set(
    string name,
    PsdConstraint[] constrs,
    double[] vals)
```

Arguments

`name`: name of double parameter.

`constrs`: a list of desired PSD constraints.

`vals`: array of values of parameter.

Model.Set()

Set values of parameter, associated with PSD constraints.

Synopsis

```
void Set(
    string name,
    PsdConstrArray constrs,
    double[] vals)
```

Arguments

`name`: name of double parameter.

`constrs`: a list of desired PSD constraints.

`vals`: array of values of parameter.

Model.SetBasis()

Set column and row basis status to model.

Synopsis

```
void SetBasis(int[] colbasis, int[] rowbasis)
```

Arguments

`colbasis`: status of column basis.

`rowbasis`: status of row basis.

Model.SetCallback()

Set user callback to COPT model.

Synopsis

```
void SetCallback(CallbackBase cb, int cbctx)
```

Arguments

`cb`: user callback instance, inheriting from CallbackBase class.

`cbctx`: COPT callback context.

Model.SetCoeff()

Set the coefficient of a variable in a linear constraint.

Synopsis

```
void SetCoeff(
    Constraint constr,
    Var var,
    double newVal)
```

Arguments

`constr`: The requested constraint.

`var`: The requested variable.

`newVal`: New coefficient.

Model.SetDblParam()

Set value of a COPT double parameter.

Synopsis

```
void SetDblParam(string param, double val)
```

Arguments

`param`: name of integer parameter.

`val`: double value.

Model.SetIntParam()

Set value of a COPT integer parameter.

Synopsis

```
void SetIntParam(string param, int val)
```

Arguments

`param`: name of integer parameter.

`val`: integer value.

Model.SetLpSolution()

Set LP solution.

Synopsis

```
void SetLpSolution(
    double[] value,
    double[] slack,
    double[] rowDual,
    double[] redCost)
```

Arguments

`value`: solution values.

`slack`: slack values.

`rowDual`: dual values.

`redCost`: reduced costs.

Model.SetMipStart()

Set initial values for variables of given number, starting from the first one.

Synopsis

```
void SetMipStart(int count, double[] vals)
```

Arguments

`count`: the number of variables to set.

`vals`: values of variables.

Model.SetMipStart()

Set initial value for the specified variable.

Synopsis

```
void SetMipStart(Var var, double val)
```

Arguments

`var`: an interested variable.

`val`: initial value of the variable.

Model.SetMipStart()

Set initial value for the specified variable.

Synopsis

```
void SetMipStart(Var[] vars, double[] vals)
```

Arguments

`vars`: a list of interested variables.

`vals`: initial values of the variables.

Model.SetMipStart()

Set initial value for the specified variable.

Synopsis

```
void SetMipStart(VarArray vars, double[] vals)
```

Arguments

`vars`: a list of interested variables.

`vals`: initial values of the variables.

Model.SetObjConst()

Set objective constant.

Synopsis

```
void SetObjConst(double constant)
```

Arguments

`constant`: constant value to set.

Model.SetObjective()

Set objective for model.

Synopsis

```
void SetObjective(Expr expr, int sense)
```

Arguments

`expr`: expression of the objective.

`sense`: optional, default value 0 does not change COPT sense

Model.SetObjSense()

Set objective sense for model.

Synopsis

```
void SetObjSense(int sense)
```

Arguments

`sense`: the objective sense.

Model.SetPsdCoeff()

Set the coefficient matrix of a PSD variable in a PSD constraint.

Synopsis

```
void SetPsdCoeff(
    PsdConstraint constr,
    PsdVar var,
    SymMatrix mat)
```

Arguments

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

`mat`: new coefficient matrix.

Model.SetPsdObjective()

Set PSD objective for model.

Synopsis

```
void SetPsdObjective(PsdExpr expr, int sense)
```

Arguments

`expr`: PSD expression of the objective.

`sense`: optional, default value 0 does not change COPT sense.

Model.SetQuadObjective()

Set quadratic objective for model.

Synopsis

```
void SetQuadObjective(QuadExpr expr, int sense)
```

Arguments

`expr`: quadratic expression of the objective.

`sense`: default value 0 does not change COPT sense.

Model.SetSlackBasis()

Set slack basis to model.

Synopsis

```
void SetSlackBasis()
```

Model.SetSolverLogFile()

Set log file for COPT.

Synopsis

```
void SetSolverLogFile(string filename)
```

Arguments

`filename`: log file name.

Model.Solve()

Solve the model as MIP.

Synopsis

```
void Solve()
```

Model.SolveLp()

Solve the model as LP.

Synopsis

```
void SolveLp()
```

Model.Tune()

Tune model.

Synopsis

```
void Tune()
```

Model.Write()

Output problem, solution, basis, MIP start or modified COPT parameters to file.

Synopsis

```
void Write(string filename)
```

Arguments

`filename`: an output file name.

Model.WriteBasis()

Output optimal basis to a file of type '.bas'.

Synopsis

```
void WriteBasis(string filename)
```

Arguments

`filename`: an output file name.

Model.WriteBin()

Output problem to a file as COPT binary format.

Synopsis

```
void WriteBin(string filename)
```

Arguments

`filename`: an output file name.

Model.WriteIIS()

Output IIS to file.

Synopsis

```
void WriteIIS(string filename)
```

Arguments

`filename`: Output file name.

Model.WriteLp()

Output problem to a file as LP format.

Synopsis

```
void WriteLp(string filename)
```

Arguments

`filename`: an output file name.

Model.WriteMps()

Output problem to a file as MPS format.

Synopsis

```
void WriteMps(string filename)
```

Arguments

`filename`: an output file name.

Model.WriteMpsStr()

Output MPS problem to problem buffer.

Synopsis

```
ProbBuffer WriteMpsStr()
```

Return

problem buffer for string of MPS problem.

Model.WriteMst()

Output MIP start information to a file of type '.mst'.

Synopsis

```
void WriteMst(string filename)
```

Arguments

`filename`: an output file name.

Model.WriteParam()

Output modified COPT parameters to a file of type '.par'.

Synopsis

```
void WriteParam(string filename)
```

Arguments

`filename`: an output file name.

Model.WritePoolSol()

Output selected pool solution to a file of type '.sol'.

Synopsis

```
void WritePoolSol(int idx, string filename)
```

Arguments

`idx`: index of pool solution.

`filename`: an output file name.

Model.WriteRelax()

Output feasibility relaxation problem to file.

Synopsis

```
void WriteRelax(string filename)
```

Arguments

`filename`: Output file name.

Model.WriteSol()

Output solution to a file of type '.sol'.

Synopsis

```
void WriteSol(string filename)
```

Arguments

`filename`: an output file name.

Model.WriteTuneParam()

Output specified tuned parameters to a file of type '.par'.

Synopsis

```
void WriteTuneParam(int idx, string filename)
```

Arguments

`idx`: Index of tuned parameters.

`filename`: Output file name.

19.2.4 Var

COPT variable object. Variables are always associated with a particular model. User creates a variable object by adding a variable to a model, rather than by using constructor of Var class.

Var.Get()

Get attribute value of the variable. Support "Value", "RedCost", "LB", "UB", and "Obj" attributes.

Synopsis

```
double Get(string attr)
```

Arguments

`attr`: attribute name.

Return

attribute value.

Var.GetBasis()

Get basis status of the variable.

Synopsis

```
int GetBasis()
```

Return

Basis status.

Var.GetIdx()

Get index of the variable.

Synopsis

```
int GetIdx()
```

Return

variable index.

Var.GetLowerIIS()

Get IIS status for lower bound of the variable.

Synopsis

```
int GetLowerIIS()
```

Return

IIS status.

Var.GetName()

Get name of the variable.

Synopsis

```
string GetName()
```

Return

variable name.

Var.GetType()

Get type of the variable.

Synopsis

```
char GetType()
```

Return

variable type.

Var.GetUpperIIS()

Get IIS status for upper bound of the variable.

Synopsis

```
int GetUpperIIS()
```

Return

IIS status.

Var.Remove()

Remove variable from model.

Synopsis

```
void Remove()
```

Var.Set()

Set attribute value of the variable. Support "LB", "UB" and "Obj" attributes.

Synopsis

```
void Set(string attr, double val)
```

Arguments

`attr`: attribute name.

`val`: new value.

Var.SetName()

Set name of the variable.

Synopsis

```
void SetName(string name)
```

Arguments

`name`: variable name.

Var.SetType()

Set type of the variable.

Synopsis

```
void SetType(char vtype)
```

Arguments

`vtype`: variable type.

19.2.5 VarArray

COPT variable array object. To store and access a set of C# *Var* objects, Cardinal Optimizer provides C# VarArray class, which defines the following methods.

VarArray.VarArray()

Constructor of vararray.

Synopsis

```
VarArray()
```

VarArray.GetVar()

Get idx-th variable object.

Synopsis

```
Var GetVar(int idx)
```

Arguments

`idx`: index of the variable.

Return

variable object with index idx.

VarArray.PushBack()

Add a variable object to variable array.

Synopsis

```
void PushBack(Var var)
```

Arguments

`var`: a variable object.

VarArray.Size()

Get the number of variable objects.

Synopsis

```
int Size()
```

Return

number of variable objects.

19.2.6 Expr

COPT linear expression object. A linear expression consists of a constant term, a list of terms of variables and associated coefficients. Linear expressions are used to build constraints.

Expr.Expr()

Constructor of a constant linear expression with default constant value 0.

Synopsis

```
Expr(double constant)
```

Arguments

`constant`: optional, constant value in expression object.

Expr.Expr()

Constructor of a linear expression with one term.

Synopsis

```
Expr(Var var, double coeff)
```

Arguments

`var`: variable for the added term.

`coeff`: coefficent for the added term with default value 1.0.

Expr.AddConstant()

Add extra constant to the expression.

Synopsis

```
void AddConstant(double constant)
```

Arguments

`constant`: delta value to be added to expression constant.

Expr.AddExpr()

Add an expression to self.

Synopsis

```
void AddExpr(Expr expr, double mult)
```

Arguments

`expr`: expression to be added.

`mult`: multiply constant.

Expr.AddTerm()

Add a term to expression object.

Synopsis

```
void AddTerm(Var var, double coeff)
```

Arguments

`var`: a variable for new term.

`coeff`: coefficient for new term.

Expr.AddTerms()

Add terms to expression object.

Synopsis

```
void AddTerms(Var[] vars, double coeff)
```

Arguments

`vars`: variables for added terms.

`coeff`: coefficient array for added terms with default value 1.0.

Expr.AddTerms()

Add terms to expression object.

Synopsis

```
void AddTerms(Var[] vars, double[] coeffs)
```

Arguments

`vars`: variables for added terms.

`coeffs`: coefficients array for added terms.

Expr.AddTerms()

Add terms to expression object.

Synopsis

```
void AddTerms(VarArray vars, double coeff)
```

Arguments

`vars`: variables for added terms.

`coeff`: coefficient array for added terms with default value 1.0.

Expr.AddTerms()

Add terms to expression object.

Synopsis

```
void AddTerms(VarArray vars, double[] coeffs)
```

Arguments

`vars`: variables for added terms.

`coeffs`: coefficients array for added terms.

Expr.Clone()

Deep copy linear expression object.

Synopsis

```
Expr Clone()
```

Return

cloned linear expression object.

Expr.GetCoeff()

Get coefficient from the i-th term in expression.

Synopsis

```
double GetCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term in expression object.

Expr.GetConstant()

Get constant in expression.

Synopsis

```
double GetConstant()
```

Return

constant in expression.

Expr.GetVar()

Get variable from the i-th term in expression.

Synopsis

```
Var GetVar(int i)
```

Arguments

`i`: index of the term.

Return

variable of the i-th term in expression object.

Expr.Remove()

Remove idx-th term from expression object.

Synopsis

```
void Remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

Expr.Remove()

Remove the term associated with variable from expression.

Synopsis

```
void Remove(Var var)
```

Arguments

`var`: a variable whose term should be removed.

Expr.SetCoeff()

Set coefficient for the i-th term in expression.

Synopsis

```
void SetCoeff(int i, double val)
```

Arguments

`i`: index of the term.

`val`: coefficient of the term.

Expr.SetConstant()

Set constant for the expression.

Synopsis

```
void SetConstant(double constant)
```

Arguments

`constant`: the value of the constant.

Expr.Size()

Get number of terms in expression.

Synopsis

```
long Size()
```

Return

number of terms.

19.2.7 Constraint

COPT constraint object. Constraints are always associated with a particular model. User creates a constraint object by adding a constraint to a model, rather than by using constructor of Constraint class.

Constraint.Get()

Get attribute value of the constraint. Support "Dual", "Slack", "LB", "UB" attributes.

Synopsis

```
double Get(string attr)
```

Arguments

`attr`: name of the attribute being queried.

Return

attribute value.

Constraint.GetBasis()

Get basis status of this constraint.

Synopsis

```
int GetBasis()
```

Return

basis status.

Constraint.GetIdx()

Get index of the constraint.

Synopsis

```
int GetIdx()
```

Return

the index of the constraint.

Constraint.GetLowerIIS()

Get IIS status for lower bound of the constraint.

Synopsis

```
int GetLowerIIS()
```

Return

IIS status.

Constraint.GetName()

Get name of the constraint.

Synopsis

```
string GetName()
```

Return

the name of the constraint.

Constraint.GetUpperIIS()

Get IIS status for upper bound of the constraint.

Synopsis

```
int GetUpperIIS()
```

Return

IIS status.

Constraint.Remove()

Remove this constraint from model.

Synopsis

```
void Remove()
```

Constraint.Set()

Set attribute value of the constraint. Support "LB" and "UB" attributes.

Synopsis

```
void Set(string attr, double val)
```

Arguments

`attr`: name of the attribute.

`val`: new value.

Constraint.SetName()

Set name for the constraint.

Synopsis

```
void SetName(string name)
```

Arguments

`name`: the name to set.

19.2.8 ConstrArray

COPT constraint array object. To store and access a set of C# *Constraint* objects, Cardinal Optimizer provides C# ConstrArray class, which defines the following methods.

ConstrArray.ConstrArray()

Constructor of constrarray object.

Synopsis

```
ConstrArray()
```

ConstrArray.GetConstr()

Get idx-th constraint object.

Synopsis

```
Constraint GetConstr(int idx)
```

Arguments

`idx`: index of the constraint.

Return

constraint object with index idx.

ConstrArray.PushBack()

Add a constraint object to constraint array.

Synopsis

```
void PushBack(Constraint constr)
```

Arguments

`constr`: a constraint object.

ConstrArray.Size()

Get the number of constraint objects.

Synopsis

```
int Size()
```

Return

number of constraint objects.

19.2.9 ConstrBuilder

COPT constraint builder object. To help building a constraint, given a linear expression, constraint sense and right-hand side value, Cardinal Optimizer provides C# ConstrBuilder class, which defines the following methods.

ConstrBuilder.ConstrBuilder()

Constructor of constrbuilder object.

Synopsis

```
ConstrBuilder()
```

ConstrBuilder.GetExpr()

Get expression associated with constraint.

Synopsis

```
Expr GetExpr()
```

Return

expression object.

ConstrBuilder.GetRange()

Get range from lower bound to upper bound of range constraint.

Synopsis

```
double GetRange()
```

Return

length from lower bound to upper bound of the constraint.

ConstrBuilder.GetSense()

Get sense associated with constraint.

Synopsis

```
char GetSense()
```

Return

constraint sense.

ConstrBuilder.Set()

Set detail of a constraint to its builder object.

Synopsis

```
void Set(
    Expr expr,
    char sense,
    double rhs)
```

Arguments

`expr`: expression object at one side of the constraint

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of the constraint.

ConstrBuilder.SetRange()

Set a range constraint to its builder.

Synopsis

```
void SetRange(Expr expr, double range)
```

Arguments

`expr`: expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must greater than 0.

19.2.10 ConstrBuilderArray

COPT constraint builder array object. To store and access a set of C# *ConstrBuilder* objects, Cardinal Optimizer provides C# ConstrBuilderArray class, which defines the following methods.

ConstrBuilderArray.ConstrBuilderArray()

Constructor of constrbuilderarray object.

Synopsis

```
ConstrBuilderArray()
```

ConstrBuilderArray.GetBuilder()

Get idx-th constraint builder object.

Synopsis

```
ConstrBuilder GetBuilder(int idx)
```

Arguments

`idx`: index of the constraint builder.

Return

constraint builder object with index idx.

ConstrBuilderArray.PushBack()

Add a constraint builder object to constraint builder array.

Synopsis

```
void PushBack(ConstrBuilder builder)
```

Arguments

`builder`: a constraint builder object.

ConstrBuilderArray.Size()

Get the number of constraint builder objects.

Synopsis

```
int Size()
```

Return

number of constraint builder objects.

19.2.11 Column

COPT column object. A column consists of a list of constraints and associated coefficients. Columns are used to represent the set of constraints in which a variable participates, and the asssociated coefficents.

Column.Column()

Constructor of column.

Synopsis

```
Column()
```

Column.AddColumn()

Add a column to self.

Synopsis

```
void AddColumn(Column col, double mult)
```

Arguments

`col`: column object to be added.

`mult`: multiply constant.

Column.AddTerm()

Add a term to column object.

Synopsis

```
void AddTerm(Constraint constr, double coeff)
```

Arguments

`constr`: a constraint for new term.

`coeff`: coefficient for new term.

Column.AddTerms()

Add terms to column object.

Synopsis

```
void AddTerms(Constraint[] constrs, double coeff)
```

Arguments

`constrs`: constraints for added terms.

`coeff`: coefficient for added terms,default value is 1.

Column.AddTerms()

Add terms to column object.

Synopsis

```
void AddTerms(Constraint[] constrs, double[] coeffs)
```

Arguments

`constrs`: constraints for added terms.

`coeffs`: coefficients for added terms.

Column.AddTerms()

Add terms to column object.

Synopsis

```
void AddTerms(ConstrArray constrs, double coeff)
```

Arguments

`constrs`: constraints for added terms.

`coeff`: coefficient for added terms,default value is 1.

Column.AddTerms()

Add terms to column object.

Synopsis

```
void AddTerms(ConstrArray constrs, double[] coeffs)
```

Arguments

`constrs`: constraints for added terms.

`coeffs`: coefficients for added terms.

Column.Clear()

Clear all terms.

Synopsis

```
void Clear()
```

Column.Clone()

Deep copy column object.

Synopsis

```
Column Clone()
```

Return

cloned column object.

Column.GetCoeff()

Get coefficient from the i-th term in column object.

Synopsis

```
double GetCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term in column object.

Column.GetConstr()

Get constraint from the i-th term in column object.

Synopsis

```
Constraint GetConstr(int i)
```

Arguments

`i`: index of the term.

Return

constraint of the i-th term in column object.

Column.Remove()

Remove idx-th term from column object.

Synopsis

```
void Remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

Column.Remove()

Remove the term associated with constraint from column object.

Synopsis

```
void Remove(Constraint constr)
```

Arguments

`constr`: a constraint whose term should be removed.

Column.Size()

Get number of terms in column object.

Synopsis

```
int Size()
```

Return

number of terms.

19.2.12 ColumnArray

COPT column array object. To store and access a set of C# *Column* objects, Cardinal Optimizer provides C# ColumnArray class, which defines the following methods.

ColumnArray.ColumnArray()

Constructor of columnarray object.

Synopsis

```
ColumnArray()
```

ColumnArray.Clear()

Clear all column objects.

Synopsis

```
void Clear()
```

ColumnArray.GetColumn()

Get idx-th column object.

Synopsis

```
Column GetColumn(int idx)
```

Arguments

`idx`: index of the column.

Return

column object with index idx.

ColumnArray.PushBack()

Add a column object to column array.

Synopsis

```
void PushBack(Column col)
```

Arguments

`col`: a column object.

ColumnArray.Size()

Get the number of column objects.

Synopsis

```
int Size()
```

Return

number of column objects.

19.2.13 Sos

COPT SOS constraint object. SOS constraints are always associated with a particular model. User creates an SOS constraint object by adding an SOS constraint to a model, rather than by using constructor of Sos class.

An SOS constraint can be type 1 or 2 (`COPT_SOS_TYPE1` or `COPT_SOS_TYPE2`).

Sos.GetIdx()

Get the index of SOS constarint.

Synopsis

```
int GetIdx()
```

Return

index of SOS constraint.

Sos.GetIIS()

Get IIS status of the SOS constraint.

Synopsis

```
int GetIIS()
```

Return

IIS status.

Sos.Remove()

Remove the SOS constraint from model.

Synopsis

```
void Remove()
```

19.2.14 SosArray

COPT SOS constraint array object. To store and access a set of C# *Sos* objects, Cardinal Optimizer provides C# SosArray class, which defines the following methods.

SosArray.SosArray()

Constructor of sosarray object.

Synopsis

```
SosArray()
```

SosArray.GetSos()

Get idx-th SOS object.

Synopsis

```
Sos GetSos(int idx)
```

Arguments

`idx`: index of SOS.

Return

SOS object with index idx.

SosArray.PushBack()

Add a SOS constraint object to SOS constraint array.

Synopsis

```
void PushBack(Sos sos)
```

Arguments

`sos`: a SOS constraint object.

SosArray.Size()

Get the number of SOS constraint objects.

Synopsis

```
int Size()
```

Return

number of SOS constraint objects.

19.2.15 SosBuilder

COPT SOS constraint builder object. To help building an SOS constraint, given the SOS type, a set of variables and associated weights, Cardinal Optimizer provides C# SosBuilder class, which defines the following methods.

SosBuilder.SosBuilder()

Constructor of sosbuilder object.

Synopsis

```
SosBuilder()
```

SosBuilder.GetSize()

Get number of terms in SOS constraint.

Synopsis

```
int GetSize()
```

Return

number of terms.

SosBuilder.GetType()

Get type of SOS constraint.

Synopsis

```
int GetType()
```

Return

type of SOS constraint.

SosBuilder.GetVar()

Get variable from the idx-th term in SOS constraint.

Synopsis

```
Var GetVar(int idx)
```

Arguments

`idx`: index of the term.

Return

variable of the idx-th term in SOS constraint.

SosBuilder.GetVars()

Get all variables in a SOS constraint.

Synopsis

```
VarArray GetVars()
```

Return

variables in a SOS constraint.

SosBuilder.GetWeight()

Get weight from the idx-th term in SOS constraint.

Synopsis

```
double GetWeight(int idx)
```

Arguments

`idx`: index of the term.

Return

weight of the idx-th term in SOS constraint.

SosBuilder.GetWeights()

Get weights of all terms in SOS constraint.

Synopsis

```
double[] GetWeights()
```

Return

array of weights.

SosBuilder.Set()

Set variables and weights of SOS constraint.

Synopsis

```
void Set(
    VarArray vars,
    double[] weights,
    int type)
```

Arguments

`vars`: variable array object.

`weights`: pointer to array of weights.

`type`: type of SOS constraint.

19.2.16 SosBuilderArray

COPT SOS constraint builder array object. To store and access a set of C# *SosBuilder* objects, Cardinal Optimizer provides C# SosBuilderArray class, which defines the following methods.

SosBuilderArray.SosBuilderArray()

Constructor of sosbuilderarray object.

Synopsis

```
SosBuilderArray()
```

SosBuilderArray.GetBuilder()

Get idx-th SOS constraint builder object.

Synopsis

```
SosBuilder GetBuilder(int idx)
```

Arguments

`idx`: index of the SOS constraint builder.

Return

SOS constraint builder object with index idx.

SosBuilderArray.PushBack()

Add a SOS constraint builder object to SOS constraint builder array.

Synopsis

```
void PushBack(SosBuilder builder)
```

Arguments

`builder`: a SOS constraint builder object.

SosBuilderArray.Size()

Get the number of SOS constraint builder objects.

Synopsis

```
int Size()
```

Return

number of SOS constraint builder objects.

19.2.17 GenConstr

COPT general constraint object. General constraints are always associated with a particular model. User creates a general constraint object by adding a general constraint to a model, rather than by using constructor of GenConstr class.

GenConstr.GetIdx()

Get the index of the general constraint.

Synopsis

```
int GetIdx()
```

Return

index of the general constraint.

GenConstr.GetIIS()

Get IIS status of the general constraint.

Synopsis

```
int GetIIS()
```

Return

IIS status.

GenConstr.Remove()

Remove the general constraint from model.

Synopsis

```
void Remove()
```

19.2.18 GenConstrArray

COPT general constraint array object. To store and access a set of C# *GenConstr* objects, Cardinal Optimizer provides C# GenConstrArray class, which defines the following methods.

GenConstrArray.GenConstrArray()

Constructor of genconstrarray.

Synopsis

```
GenConstrArray()
```

GenConstrArray.GetGenConstr()

Get idx-th general constraint object.

Synopsis

```
GenConstr GetGenConstr(int idx)
```

Arguments

`idx`: index of the general constraint.

Return

general constraint object with index idx.

GenConstrArray.PushBack()

Add a general constraint object to general constraint array.

Synopsis

```
void PushBack(GenConstr genconstr)
```

Arguments

`genconstr`: a general constraint object.

GenConstrArray.Size()

Get the number of general constraint objects.

Synopsis

```
int Size()
```

Return

number of general constraint objects.

19.2.19 GenConstrBuilder

COPT general constraint builder object. To help building a general constraint, given a binary variable and associated value, a linear expression and constraint sense, Cardinal Optimizer provides C# GenConstrBuilder class, which defines the following methods.

GenConstrBuilder.GenConstrBuilder()

Constructor of genconstrbuilder.

Synopsis

```
GenConstrBuilder()
```

GenConstrBuilder.GetBinVal()

Get binary value associated with general constraint.

Synopsis

```
int GetBinVal()
```

Return

binary value.

GenConstrBuilder.GetBinVar()

Get binary variable associated with general constraint.

Synopsis

```
Var GetBinVar()
```

Return

binary vaiable object.

GenConstrBuilder.GetExpr()

Get expression associated with general constraint.

Synopsis

```
Expr GetExpr()
```

Return

expression object.

GenConstrBuilder.GetSense()

Get sense associated with general constraint.

Synopsis

```
char GetSense()
```

Return

constraint sense.

GenConstrBuilder.Set()

Set binary variable, binary value, expression and sense of general constraint.

Synopsis

```
void Set(
    Var binvar,
    int binval,
    Expr expr,
    char sense)
```

Arguments

`binvar`: binary variable.

`binval`: binary value.

`expr`: expression object.

`sense`: general constraint sense.

19.2.20 GenConstrBuilderArray

COPT general constraint builder array object. To store and access a set of C# *GenConstrBuilder* objects, Cardinal Optimizer provides C# GenConstrBuilderArray class, which defines the following methods.

GenConstrBuilderArray.GenConstrBuilderArray()

Constructor of genconstrbuilderarray.

Synopsis

```
GenConstrBuilderArray()
```

GenConstrBuilderArray.GetBuilder()

Get idx-th general constraint builder object.

Synopsis

```
GenConstrBuilder GetBuilder(int idx)
```

Arguments

`idx`: index of the general constraint builder.

Return

general constraint builder object with index idx.

GenConstrBuilderArray.PushBack()

Add a general constraint builder object to general constraint builder array.

Synopsis

```
void PushBack(GenConstrBuilder builder)
```

Arguments

`builder`: a general constraint builder object.

GenConstrBuilderArray.Size()

Get the number of general constraint builder objects.

Synopsis

```
int Size()
```

Return

number of general constraint builder objects.

19.2.21 Cone

COPT cone constraint object. Cone constraints are always associated with a particular model. User creates a cone constraint object by adding a cone constraint to a model, rather than by using constructor of Cone class.

A cone constraint can be regular or rotated (`COPT_CONE_QUAD` or `COPT_CONE_RQUAD`).

Cone.GetIdx()

Get the index of a cone constarint.

Synopsis

```
int GetIdx()
```

Return

index of the cone constraint.

Cone.Remove()

Remove the cone constraint from model.

Synopsis

```
void Remove()
```

19.2.22 ConeArray

COPT cone constraint array object. To store and access a set of C# *Cone* objects, Cardinal Optimizer provides C# ConeArray class, which defines the following methods.

ConeArray.ConeArray()

Constructor of conearray object.

Synopsis

```
ConeArray()
```

ConeArray.GetCone()

Get idx-th cone object.

Synopsis

```
Cone GetCone(int idx)
```

Arguments

`idx`: index of cone.

Return

cone object with index idx.

ConeArray.PushBack()

Add a cone constraint object to cone constraint array.

Synopsis

```
void PushBack(Cone cone)
```

Arguments

`cone`: a cone constraint object.

ConeArray.Size()

Get the number of cone constraint objects.

Synopsis

```
int Size()
```

Return

number of cone constraint objects.

19.2.23 ConeBuilder

COPT cone constraint builder object. To help building a cone constraint, given the cone type and a set of variables, Cardinal Optimizer provides C# ConeBuilder class, which defines the following methods.

ConeBuilder.ConeBuilder()

Constructor of conebuilder object.

Synopsis

```
ConeBuilder()
```

ConeBuilder.GetSize()

Get number of variables in a cone constraint.

Synopsis

```
int GetSize()
```

Return

number of variables.

ConeBuilder.GetType()

Get type of a cone constraint.

Synopsis

```
int GetType()
```

Return

type of the cone constraint.

ConeBuilder.GetVar()

Get i-th variable in a cone constraint.

Synopsis

```
Var GetVar(int idx)
```

Arguments

`idx`: index of vars.

Return

the i-th variable in a cone constraint.

ConeBuilder.GetVars()

Get all variables in a cone constraint.

Synopsis

```
VarArray GetVars()
```

Return

variables in a cone constraint.

ConeBuilder.Set()

Set variables and type of a cone constraint.

Synopsis

```
void Set(VarArray vars, int type)
```

Arguments

`vars`: variable array object.

`type`: type of a cone constraint.

19.2.24 ConeBuilderArray

COPT cone constraint builder array object. To store and access a set of C# *ConeBuilder* objects, Cardinal Optimizer provides C# ConeBuilderArray class, which defines the following methods.

ConeBuilderArray.ConeBuilderArray()

Constructor of conebuilderarray object.

Synopsis

```
ConeBuilderArray()
```

ConeBuilderArray.GetBuilder()

Get idx-th cone constraint builder object.

Synopsis

```
ConeBuilder GetBuilder(int idx)
```

Arguments

`idx`: index of the cone constraint builder.

Return

cone constraint builder object with index idx.

ConeBuilderArray.PushBack()

Add a cone constraint builder object to cone constraint builder array.

Synopsis

```
void PushBack(ConeBuilder builder)
```

Arguments

`builder`: a cone constraint builder object.

ConeBuilderArray.Size()

Get the number of cone constraint builder objects.

Synopsis

```
int Size()
```

Return

number of cone constraint builder objects.

19.2.25 QuadExpr

COPT quadratic expression object. A quadratic expression consists of a linear expression, a list of variable pairs and associated coefficients of quadratic terms. Quadratic expressions are used to build quadratic constraints and objectives.

QuadExpr.QuadExpr()

Constructor of a quadratic expression with default constant value 0.

Synopsis

```
QuadExpr(double constant)
```

Arguments

`constant`: optional, constant value in quadratic expression object.

QuadExpr.QuadExpr()

Constructor of a quadratic expression with one linear term.

Synopsis

```
QuadExpr(Var var, double coeff)
```

Arguments

`var`: variable of the added linear term.

`coeff`: coefficent for the added linear term with default value 1.0.

QuadExpr.QuadExpr()

Constructor of a quadratic expression with a linear expression.

Synopsis

```
QuadExpr(Expr expr)
```

Arguments

`expr`: linear expression added to the quadratic expression.

QuadExpr.QuadExpr()

Constructor of a quadratic expression with two linear expression.

Synopsis

```
QuadExpr(Expr expr, Var var)
```

Arguments

`expr`: one linear expression.

`var`: another variable.

QuadExpr.QuadExpr()

Constructor of a quadratic expression with two linear expression.

Synopsis

```
QuadExpr(Expr left, Expr right)
```

Arguments

`left`: one linear expression.

`right`: another linear expression.

QuadExpr.AddConstant()

Add a constant to the quadratic expression.

Synopsis

```
void AddConstant(double constant)
```

Arguments

`constant`: value to be added.

QuadExpr.AddLinExpr()

Add a linear expression to self.

Synopsis

```
void AddLinExpr(Expr expr)
```

Arguments

`expr`: linear expression to be added.

QuadExpr.AddLinExpr()

Add a linear expression to self.

Synopsis

```
void AddLinExpr(Expr expr, double mult)
```

Arguments

`expr`: linear expression to be added.

`mult`: multiplier constant.

QuadExpr.AddQuadExpr()

Add a quadratic expression to self.

Synopsis

```
void AddQuadExpr(QuadExpr expr)
```

Arguments

`expr`: quadratic expression to be added.

QuadExpr.AddQuadExpr()

Add a quadratic expression to self.

Synopsis

```
void AddQuadExpr(QuadExpr expr, double mult)
```

Arguments

`expr`: quadratic expression to be added.

`mult`: multiplier constant.

QuadExpr.AddTerm()

Add a term to quadratic expression object.

Synopsis

```
void AddTerm(Var var, double coeff)
```

Arguments

`var`: a variable of new term.

`coeff`: coefficient of new term.

QuadExpr.AddTerm()

Add a quadratic term to expression object.

Synopsis

```
void AddTerm(
    Var var1,
    Var var2,
    double coeff)
```

Arguments

`var1`: first variable of new quadratic term.

`var2`: second variable of new quadratic term.

`coeff`: coefficient of new quadratic term.

QuadExpr.AddTerms()

Add linear terms to quadratic expression object.

Synopsis

```
void AddTerms(Var[] vars, double coeff)
```

Arguments

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

QuadExpr.AddTerms()

Add linear terms to quadratic expression object.

Synopsis

```
void AddTerms(Var[] vars, double[] coeffs)
```

Arguments

`vars`: variables of added linear terms.

`coeffs`: coefficients of added linear terms.

QuadExpr.AddTerms()

Add linear terms to quadratic expression object.

Synopsis

```
void AddTerms(VarArray vars, double coeff)
```

Arguments

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

QuadExpr.AddTerms()

Add linear terms to quadratic expression object.

Synopsis

```
void AddTerms(VarArray vars, double[] coeffs)
```

Arguments

`vars`: variables of added terms.

`coeffs`: coefficients of added terms.

QuadExpr.AddTerms()

Add quadratic terms to expression object.

Synopsis

```
void AddTerms(
    VarArray vars1,
    VarArray vars2,
    double[] coeffs)
```

Arguments

`vars1`: first set of variables for added quadratic terms.

`vars2`: second set of variables for added quadratic terms.

`coeffs`: coefficient array for added quadratic terms.

QuadExpr.AddTerms()

Add quadratic terms to expression object.

Synopsis

```
void AddTerms(
    Var[] vars1,
    Var[] vars2,
    double[] coeffs)
```

Arguments

`vars1`: first set of variables for added quadratic terms.

`vars2`: second set of variables for added quadratic terms.

`coeffs`: coefficient array for added quadratic terms.

QuadExpr.Clone()

Deep copy quadratic expression object.

Synopsis

```
QuadExpr Clone()
```

Return

cloned quadratic expression object.

QuadExpr.Evaluate()

evaluate quadratic expression after solving

Synopsis

```
double Evaluate()
```

Return

value of quadratic expression

QuadExpr.GetCoeff()

Get coefficient from the i-th term in quadratic expression.

Synopsis

```
double GetCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term in quadratic expression object.

QuadExpr.GetConstant()

Get constant in quadratic expression.

Synopsis

```
double GetConstant()
```

Return

constant in quadratic expression.

QuadExpr.GetLinExpr()

Get linear expression in quadratic expression.

Synopsis

```
Expr GetLinExpr()
```

Return

linear expression object.

QuadExpr.GetVar1()

Get first variable from the i-th term in quadratic expression.

Synopsis

```
Var GetVar1(int i)
```

Arguments

`i`: index of the term.

Return

first variable of the i-th term in quadratic expression object.

QuadExpr.GetVar2()

Get second variable from the i-th term in quadratic expression.

Synopsis

```
Var GetVar2(int i)
```

Arguments

`i`: index of the term.

Return

second variable of the i-th term in quadratic expression object.

QuadExpr.Remove()

Remove idx-th term from quadratic expression object.

Synopsis

```
void Remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

QuadExpr.Remove()

Remove the term associated with variable from quadratic expression.

Synopsis

```
void Remove(Var var)
```

Arguments

`var`: a variable whose term should be removed.

QuadExpr.SetCoeff()

Set coefficient of the i-th term in quadratic expression.

Synopsis

```
void SetCoeff(int i, double val)
```

Arguments

`i`: index of the quadratic term.

`val`: coefficient of the term.

QuadExpr.SetConstant()

Set constant for the quadratic expression.

Synopsis

```
void SetConstant(double constant)
```

Arguments

`constant`: the value of the constant.

QuadExpr.Size()

Get number of terms in quadratic expression.

Synopsis

```
long Size()
```

Return

number of quadratic terms.

19.2.26 QConstraint

COPT quadratic constraint object. Quadratic constraints are always associated with a particular model. User creates a quadratic constraint object by adding a quadratic constraint to a model, rather than by using constructor of QConstraint class.

QConstraint.Get()

Get attribute value of the quadratic constraint.

Synopsis

```
double Get(string attr)
```

Arguments

`attr`: name of the attribute being queried.

Return

attribute value.

QConstraint.GetIdx()

Get index of the quadratic constraint.

Synopsis

```
int GetIdx()
```

Return

the index of the quadratic constraint.

QConstraint.GetName()

Get name of the quadratic constraint.

Synopsis

```
string GetName()
```

Return

the name of the quadratic constraint.

QConstraint.GetRhs()

Get rhs of quadratic constraint.

Synopsis

```
double GetRhs()
```

Return

rhs of quadratic constraint.

QConstraint.GetSense()

Get rhs of quadratic constraint.

Synopsis

```
char GetSense()
```

Return

rhs of quadratic constraint.

QConstraint.Remove()

Remove this constraint from model.

Synopsis

```
void Remove()
```

QConstraint.Set()

Set attribute value of the quadratic constraint.

Synopsis

```
void Set(string attr, double val)
```

Arguments

`attr`: name of the attribute.

`val`: new value.

QConstraint.SetName()

Set name of quadratic constraint.

Synopsis

```
void SetName(string name)
```

Arguments

`name`: the name to set.

QConstraint.SetRhs()

Set rhs of quadratic constraint.

Synopsis

```
void SetRhs(double rhs)
```

Arguments

`rhs`: rhs of quadratic constraint.

QConstraint.SetSense()

Set sense of quadratic constraint.

Synopsis

```
void SetSense(char sense)
```

Arguments

`sense`: sense of quadratic constraint.

19.2.27 QConstrArray

COPT quadratic constraint array object. To store and access a set of C# *QConstraint* objects, Cardinal Optimizer provides C# QConstrArray class, which defines the following methods.

QConstrArray.QConstrArray()

Constructor of qconstrarray object.

Synopsis

```
QConstrArray()
```

QConstrArray.GetQConstr()

Get idx-th quadratic constraint object.

Synopsis

```
QConstraint GetQConstr(int idx)
```

Arguments

`idx`: index of the quadratic constraint.

Return

constraint object with index 'idx'.

QConstrArray.PushBack()

Add a quadratic constraint object to array.

Synopsis

```
void PushBack(QConstraint constr)
```

Arguments

`constr`: a quadratic constraint object.

QConstrArray.Size()

Get the number of quadratic constraint objects.

Synopsis

```
int Size()
```

Return

number of quadratic constraint objects.

19.2.28 QConstrBuilder

COPT quadratic constraint builder object. To help building a quadratic constraint, given a quadratic expression, constraint sense and right-hand side value, Cardinal Optimizer provides C# ConeBuilder class, which defines the following methods.

QConstrBuilder.QConstrBuilder()

Constructor of qconstrbuilder object.

Synopsis

```
QConstrBuilder()
```

QConstrBuilder.GetQuadExpr()

Get quadratic expression associated with constraint.

Synopsis

```
QuadExpr GetQuadExpr()
```

Return

quadratic expression object.

QConstrBuilder.GetSense()

Get sense associated with quadratic constraint.

Synopsis

```
char GetSense()
```

Return

quadratic constraint sense.

QConstrBuilder.Set()

Set detail of a quadratic constraint to its builder object.

Synopsis

```
void Set(
    QuadExpr expr,
    char sense,
    double rhs)
```

Arguments

`expr`: expression object at one side of the quadratic constraint.

`sense`: quadratic constraint sense.

`rhs`: constant of right side of quadratic constraint.

19.2.29 QConstrBuilderArray

COPT quadratic constraint builder array object. To store and access a set of C# *QConstrBuilder* objects, Cardinal Optimizer provides C# QConstrBuilderArray class, which defines the following methods.

QConstrBuilderArray.QConstrBuilderArray()

QConstructor of constrbuilderarray object.

Synopsis

```
QConstrBuilderArray()
```

QConstrBuilderArray.GetBuilder()

Get idx-th quadratic constraint builder object.

Synopsis

```
QConstrBuilder GetBuilder(int idx)
```

Arguments

`idx`: index of the quadratic constraint builder.

Return

constraint builder object with index 'idx'.

QConstrBuilderArray.PushBack()

Add a quadratic constraint builder object to constraint builder array.

Synopsis

```
void PushBack(QConstrBuilder builder)
```

Arguments

`builder`: a quadratic constraint builder object.

QConstrBuilderArray.Size()

Get the number of quadratic constraint builder objects.

Synopsis

```
int Size()
```

Return

number of quadratic constraint builder objects.

19.2.30 PsdVar

COPT PSD variable object. PSD variables are always associated with a particular model. User creates a PSD variable object by adding a PSD variable to model, rather than by constructor of PsdVar class.

PsdVar.Get()

Get attribute values of PSD variable.

Synopsis

```
double[] Get(string attr)
```

Arguments

`attr`: attribute name.

Return

output array of attribute values.

PsdVar.GetDim()

Get dimension of PSD variable.

Synopsis

```
int GetDim()
```

Return

dimension of PSD variable.

PsdVar.GetIdx()

Get length of PSD variable.

Synopsis

```
int GetIdx()
```

Return

length of PSD variable.

PsdVar.GetLen()

Get length of PSD variable.

Synopsis

```
int GetLen()
```

Return

length of PSD variable.

PsdVar.GetName()

Get name of PSD variable.

Synopsis

```
string GetName()
```

Return

name of PSD variable.

PsdVar.Remove()

Remove PSD variable from model.

Synopsis

```
void Remove()
```

19.2.31 PsdVarArray

COPT PSD variable array object. To store and access a set of *PsdVar* objects, Cardinal Optimizer provides PsdVarArray class, which defines the following methods.

PsdVarArray.PsdVarArray()

Constructor of PsdVarArray.

Synopsis

```
PsdVarArray()
```

PsdVarArray.GetPsdVar()

Get idx-th PSD variable object.

Synopsis

```
PsdVar GetPsdVar(int idx)
```

Arguments

`idx`: index of the PSD variable.

Return

PSD variable object with index idx.

PsdVarArray.PushBack()

Add a PSD variable object to PSD variable array.

Synopsis

```
void PushBack(PsdVar var)
```

Arguments

`var`: a PSD variable object.

PsdVarArray.Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for PSD variable object.

PsdVarArray.Size()

Get the number of PSD variable objects.

Synopsis

```
int Size()
```

Return

number of PSD variable objects.

19.2.32 PsdExpr

COPT PSD expression object. A PSD expression consists of a linear expression, a list of PSD variables and associated coefficient matrices of PSD terms. PSD expressions are used to build PSD constraints and objectives.

PsdExpr.PsdExpr()

Constructor of a PSD expression with default constant value 0.

Synopsis

```
PsdExpr(double constant)
```

Arguments

`constant`: optional, constant value in PSD expression object.

PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

Synopsis

```
PsdExpr(Var var, double coeff)
```

Arguments

`var`: variable for the added term.

`coeff`: coefficent for the added term.

PsdExpr.PsdExpr()

Constructor of a PSD expression with a linear expression.

Synopsis

```
PsdExpr(Expr expr)
```

Arguments

`expr`: input linear expression.

PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

Synopsis

```
PsdExpr(PsdVar var, SymMatrix mat)
```

Arguments

`var`: PSD variable for the added term.

`mat`: coefficient matrix for the added term.

PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

Synopsis

```
PsdExpr(PsdVar var, SymMatExpr expr)
```

Arguments

`var`: PSD variable for the added term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

PsdExpr.AddConstant()

Add constant to the PSD expression.

Synopsis

```
void AddConstant(double constant)
```

Arguments

`constant`: value to be added.

PsdExpr.AddLinExpr()

Add a linear expression to PSD expression object.

Synopsis

```
void AddLinExpr(Expr expr)
```

Arguments

`expr`: linear expression to be added.

PsdExpr.AddLinExpr()

Add a linear expression to PSD expression object.

Synopsis

```
void AddLinExpr(Expr expr, double mult)
```

Arguments

`expr`: linear expression to be added.

`mult`: multiplier constant.

PsdExpr.AddPsdExpr()

Add a PSD expression to self.

Synopsis

```
void AddPsdExpr(PsdExpr expr)
```

Arguments

`expr`: PSD expression to be added.

PsdExpr.AddPsdExpr()

Add a PSD expression to self.

Synopsis

```
void AddPsdExpr(PsdExpr expr, double mult)
```

Arguments

`expr`: PSD expression to be added.

`mult`: multiplier constant.

PsdExpr.AddTerm()

Add a linear term to PSD expression object.

Synopsis

```
void AddTerm(Var var, double coeff)
```

Arguments

`var`: variable of new linear term.

`coeff`: coefficient of new linear term.

PsdExpr.AddTerm()

Add a PSD term to PSD expression object.

Synopsis

```
void AddTerm(PsdVar var, SymMatrix mat)
```

Arguments

`var`: PSD variable of new PSD term.

`mat`: coefficient matrix of new PSD term.

PsdExpr.AddTerm()

Add a PSD term to PSD expression object.

Synopsis

```
void AddTerm(PsdVar var, SymMatExpr expr)
```

Arguments

`var`: PSD variable of new PSD term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

PsdExpr.AddTerms()

Add linear terms to PSD expression object.

Synopsis

```
void AddTerms(Var[] vars, double coeff)
```

Arguments

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

PsdExpr.AddTerms()

Add linear terms to PSD expression object.

Synopsis

```
void AddTerms(Var[] vars, double[] coeffs)
```

Arguments

`vars`: variables for added linear terms.

`coeffs`: coefficient array for added linear terms.

PsdExpr.AddTerms()

Add linear terms to PSD expression object.

Synopsis

```
void AddTerms(VarArray vars, double coeff)
```

Arguments

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

PsdExpr.AddTerms()

Add linear terms to PSD expression object.

Synopsis

```
void AddTerms(VarArray vars, double[] coeffs)
```

Arguments

`vars`: variables of added terms.

`coeffs`: coefficients of added terms.

PsdExpr.AddTerms()

Add PSD terms to PSD expression object.

Synopsis

```
void AddTerms(PsdVarArray vars, SymMatrixArray mats)
```

Arguments

`vars`: PSD variables for added PSD terms.

`mats`: coefficient matrixes for added PSD terms.

PsdExpr.AddTerms()

Add PSD terms to PSD expression object.

Synopsis

```
void AddTerms(PsdVar[] vars, SymMatrix[] mats)
```

Arguments

`vars`: PSD variables for added PSD terms.

`mats`: coefficient matrixes for added PSD terms.

PsdExpr.Clone()

Deep copy PSD expression object.

Synopsis

```
PsdExpr Clone()
```

Return

cloned PSD expression object.

PsdExpr.Evaluate()

evaluate PSD expression after solving

Synopsis

```
double Evaluate()
```

Return

value of PSD expression

PsdExpr.GetCoeff()

Get coefficient from the i-th term in PSD expression.

Synopsis

```
SymMatExpr GetCoeff(int i)
```

Arguments

`i`: index of the PSD term.

Return

coefficient expression of the i-th PSD term.

PsdExpr.GetConstant()

Get constant in PSD expression.

Synopsis

```
double GetConstant()
```

Return

constant in PSD expression.

PsdExpr.GetLinExpr()

Get linear expression in PSD expression.

Synopsis

```
Expr GetLinExpr()
```

Return

linear expression object.

PsdExpr.GetPsdVar()

Get the PSD variable from the i-th term in PSD expression.

Synopsis

```
PsdVar GetPsdVar(int i)
```

Arguments

`i`: index of the term.

Return

the first variable of the i-th term in PSD expression object.

PsdExpr.Multiply()

multiply a PSD expression and a constant.

Synopsis

```
void Multiply(double c)
```

Arguments

`c`: constant operand.

PsdExpr.Remove()

Remove i-th term from PSD expression object.

Synopsis

```
void Remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

PsdExpr.Remove()

Remove the term associated with variable from PSD expression.

Synopsis

```
void Remove(Var var)
```

Arguments

`var`: a variable whose term should be removed.

PsdExpr.Remove()

Remove the term associated with PSD variable from PSD expression.

Synopsis

```
void Remove(PsdVar var)
```

Arguments

`var`: a PSD variable whose term should be removed.

PsdExpr.SetCoeff()

Set coefficient matrix of the i-th term in PSD expression.

Synopsis

```
void SetCoeff(int i, SymMatrix mat)
```

Arguments

`i`: index of the PSD term.

`mat`: coefficient matrix of the term.

PsdExpr.SetConstant()

Set constant for the PSD expression.

Synopsis

```
void SetConstant(double constant)
```

Arguments

`constant`: the value of the constant.

PsdExpr.Size()

Get number of PSD terms in expression.

Synopsis

```
long Size()
```

Return

number of PSD terms.

19.2.33 PsdConstraint

COPT PSD constraint object. PSD constraints are always associated with a particular model. User creates a PSD constraint object by adding a PSD constraint to model, rather than by constructor of PsdConstraint class.

PsdConstraint.Get()

Get attribute value of the PSD constraint. Support related PSD attributes.

Synopsis

```
double Get(string attr)
```

Arguments

`attr`: name of queried attribute.

Return

attribute value.

PsdConstraint.GetIdx()

Get index of the PSD constraint.

Synopsis

```
int GetIdx()
```

Return

the index of the PSD constraint.

PsdConstraint.GetName()

Get name of the PSD constraint.

Synopsis

```
string GetName()
```

Return

the name of the PSD constraint.

PsdConstraint.Remove()

Remove this PSD constraint from model.

Synopsis

```
void Remove()
```

PsdConstraint.Set()

Set attribute value of the PSD constraint. Support related PSD attributes.

Synopsis

```
void Set(string attr, double value)
```

Arguments

`attr`: name of queried attribute.

`value`: new value.

PsdConstraint.SetName()

Set name of a PSD constraint.

Synopsis

```
void SetName(string name)
```

Arguments

`name`: the name to set.

19.2.34 PsdConstrArray

COPT PSD constraint array object. To store and access a set of *PsdConstraint* objects, Cardinal Optimizer provides PsdConstrArray class, which defines the following methods.

PsdConstrArray.PsdConstrArray()

Constructor of PsdConstrArray object.

Synopsis

```
PsdConstrArray()
```

PsdConstrArray.GetPsdConstr()

Get idx-th PSD constraint object.

Synopsis

```
PsdConstraint GetPsdConstr(int idx)
```

Arguments

`idx`: index of the PSD constraint.

Return

PSD constraint object with index idx.

PsdConstrArray.PushBack()

Add a PSD constraint object to PSD constraint array.

Synopsis

```
void PushBack(PsdConstraint constr)
```

Arguments

`constr`: a PSD constraint object.

PsdConstrArray.Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for PSD constraint objects.

PsdConstrArray.Size()

Get the number of PSD constraint objects.

Synopsis

```
int Size()
```

Return

number of PSD constraint objects.

19.2.35 PsdConstrBuilder

COPT PSD constraint builder object. To help building a PSD constraint, given a PSD expression, constraint sense and right-hand side value, Cardinal Optimizer provides PsdConstrBuilder class, which defines the following methods.

PsdConstrBuilder.PsdConstrBuilder()

Constructor of PsdConstrBuilder object.

Synopsis

```
PsdConstrBuilder()
```

PsdConstrBuilder.GetPsdExpr()

Get expression associated with PSD constraint.

Synopsis

```
PsdExpr GetPsdExpr()
```

Return

PSD expression object.

PsdConstrBuilder.GetRange()

Get range from lower bound to upper bound of range constraint.

Synopsis

```
double GetRange()
```

Return

length from lower bound to upper bound of the constraint.

PsdConstrBuilder.GetSense()

Get sense associated with PSD constraint.

Synopsis

```
char GetSense()
```

Return

PSD constraint sense.

PsdConstrBuilder.Set()

Set detail of a PSD constraint to its builder object.

Synopsis

```
void Set(
    PsdExpr expr,
    char sense,
    double rhs)
```

Arguments

`expr`: expression object at one side of the PSD constraint.

`sense`: PSD constraint sense, other than COPT_RANGE.

`rhs`: constant at right side of the PSD constraint.

PsdConstrBuilder.SetRange()

Set a range constraint to its builder.

Synopsis

```
void SetRange(PsdExpr expr, double range)
```

Arguments

`expr`: PSD expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must greater than 0.

19.2.36 PsdConstrBuilderArray

COPT PSD constraint builder array object. To store and access a set of *PsdConstrBuilder* objects, Cardinal Optimizer provides PsdConstrBuilderArray class, which defines the following methods.

PsdConstrBuilderArray.PsdConstrBuilderArray()

Constructor of PsdConstrBuilderArray object.

Synopsis

```
PsdConstrBuilderArray()
```

PsdConstrBuilderArray.GetBuilder()

Get idx-th PSD constraint builder object.

Synopsis

```
PsdConstrBuilder GetBuilder(int idx)
```

Arguments

`idx`: index of the PSD constraint builder.

Return

PSD constraint builder object with index idx.

PsdConstrBuilderArray.PushBack()

Add a PSD constraint builder object to PSD constraint builder array.

Synopsis

```
void PushBack(PsdConstrBuilder builder)
```

Arguments

`builder`: a PSD constraint builder object.

PsdConstrBuilderArray.Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for PSD constraint builder object.

PsdConstrBuilderArray.Size()

Get the number of PSD constraint builder objects.

Synopsis

```
int Size()
```

Return

number of PSD constraint builder objects.

19.2.37 SymMatrix

COPT symmetric matrix object. Symmetric matrices are always associated with a particular model. User creates a symmetric matrix object by adding a symmetric matrix to model, rather than by constructor of SymMatrix class.

Symmetric matrices are used as coefficient matrices of PSD terms in PSD expressions, PSD constraints or PSD objectives.

SymMatrix.GetDim()

Get the dimension of a symmetric matrix.

Synopsis

```
int GetDim()
```

Return

dimension of a symmetric matrix.

SymMatrix.GetIdx()

Get the index of a symmetric matrix.

Synopsis

```
int GetIdx()
```

Return

index of a symmetric matrix.

19.2.38 SymMatrixArray

COPT symmetric matrix object. To store and access a set of *SymMatrix* objects, Cardinal Optimizer provides SymMatrixArray class, which defines the following methods.

SymMatrixArray.SymMatrixArray()

Constructor of SymMatrixAarray.

Synopsis

```
SymMatrixArray()
```

SymMatrixArray.GetMatrix()

Get i-th SymMatrix object.

Synopsis

```
SymMatrix GetMatrix(int idx)
```

Arguments

`idx`: index of the SymMatrix object.

Return

SymMatrix object with index idx.

SymMatrixArray.PushBack()

Add a SymMatrix object to SymMatrix array.

Synopsis

```
void PushBack(SymMatrix mat)
```

Arguments

`mat`: a SymMatrix object.

SymMatrixArray.Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for symmetric matrix object.

SymMatrixArray.Size()

Get the number of SymMatrix objects.

Synopsis

```
int Size()
```

Return

number of SymMatrix objects.

19.2.39 SymMatExpr

COPT symmetric matrix expression object. A symmetric matrix expression is a linear combination of symmetric matrices, which is still a symmetric matrix. However, by doing so, we are able to delay computing the final matrix until setting PSD constraints or PSD objective.

SymMatExpr.SymMatExpr()

Constructor of a symmetric matrix expression.

Synopsis

```
SymMatExpr()
```

SymMatExpr.SymMatExpr()

Constructor of a symmetric matrix expression with one term.

Synopsis

```
SymMatExpr(SymMatrix mat, double coeff)
```

Arguments

`mat`: symmetric matrix of the added term.

`coeff`: optional, coefficent for the added term. Its default value is 1.0.

SymMatExpr.AddSymMatExpr()

Add a symmetric matrix expression to self.

Synopsis

```
void AddSymMatExpr(SymMatExpr expr, double mult)
```

Arguments

`expr`: symmetric matrix expression to be added.

`mult`: optional, constant multiplier, default value is 1.0.

SymMatExpr.AddTerm()

Add a term to symmetric matrix expression object.

Synopsis

```
bool AddTerm(SymMatrix mat, double coeff)
```

Arguments

`mat`: symmetric matrix of the new term.

`coeff`: coefficient of the new term.

Return

True if the term is added successfully.

SymMatExpr.AddTerms()

Add multiple terms to expression object.

Synopsis

```
int AddTerms(SymMatrixArray mats, double[] coeffs)
```

Arguments

`mats`: symmetric matrix array object for added terms.

`coeffs`: coefficient array for added terms.

Return

Number of added terms. If negative, fail to add one of terms.

SymMatExpr.AddTerms()

Add multiple terms to expression object.

Synopsis

```
int AddTerms(SymMatrix[] mats, double[] coeffs)
```

Arguments

`mats`: symmetric matrix array object for added terms.

`coeffs`: coefficient array for added terms.

Return

Number of added terms. If negative, fail to add one of terms.

SymMatExpr.AddTerms()

Add multiple terms to expression object.

Synopsis

```
int AddTerms(SymMatrix[] mats, double coeff)
```

Arguments

`mats`: symmetric matrix array object for added terms.

`coeff`: optional, common coefficient for added terms, default is 1.0.

Return

Number of added terms. If negative, fail to add one of terms.

SymMatExpr.Clone()

Deep copy symmetric matrix expression object.

Synopsis

```
SymMatExpr Clone()
```

Return

cloned expression object.

SymMatExpr.GetCoeff()

Get coefficient of the i-th term in expression object.

Synopsis

```
double GetCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term.

SymMatExpr.GetDim()

Get dimension of symmetric matrix in expression.

Synopsis

```
int GetDim()
```

Return

dimension of symmetric matrix.

SymMatExpr.GetSymMat()

Get symmetric matrix of the i-th term in expression object.

Synopsis

```
SymMatrix GetSymMat(int i)
```

Arguments

`i`: index of the term.

Return

the symmetric matrix of the i-th term.

SymMatExpr.Multiply()

multiply a symmetric matrix expression and a constant.

Synopsis

```
void Multiply(double c)
```

Arguments

`c`: constant operand.

SymMatExpr.Remove()

Remove i-th term from expression object.

Synopsis

```
void Remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

SymMatExpr.Remove()

Remove the term associated with the symmetric matrix.

Synopsis

```
void Remove(SymMatrix mat)
```

Arguments

`mat`: a symmetric matrix whose term should be removed.

SymMatExpr.Reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void Reserve(int n)
```

Arguments

`n`: minimum capacity for expression object.

SymMatExpr.SetCoeff()

Set coefficient for the i-th term in expression object.

Synopsis

```
void SetCoeff(int i, double val)
```

Arguments

`i`: index of the term.

`val`: coefficient of the term.

SymMatExpr.Size()

Get number of terms in expression.

Synopsis

```
long Size()
```

Return

number of terms.

19.2.40 CallbackBase

COPT Callback abstract base object. Users must implment its virtual method `virtual void CallbackBase::callback()` to instantiate an instance, which pass to `Model::SetCallback(CallbackBase cb, int cbctx)` as the first parameter. Subclass of CallbackBase inherits the following member methods:

CallbackBase.CallbackBase()

Constructor of CallbackBase, implementing ICallback interface.

Synopsis

```
CallbackBase()
```

CallbackBase.AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(
    Expr lhs,
    char sense,
    Expr rhs)
```

Arguments

`lhs`: left hand side expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy contraint.

CallbackBase.AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(ConstrBuilder builder)
```

Arguments

`builder`: builder for lazy contraint.

CallbackBase.AddLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void AddLazyConstr(
    Expr lhs,
    char sense,
    double rhs)
```

Arguments

`lhs`: expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

CallbackBase.AddLazyConstrs()

Add lazy constraints to model.

Synopsis

```
void AddLazyConstrs(ConstrBuilderArray builders)
```

Arguments

`builders`: array of builders for lazy contraints.

CallbackBase.AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(
    Expr lhs,
    char sense,
    double rhs)
```

Arguments

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

CallbackBase.AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(
    Expr lhs,
    char sense,
    Expr rhs)
```

Arguments

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

CallbackBase.AddUserCut()

Add a user cut to model.

Synopsis

```
void AddUserCut(ConstrBuilder builder)
```

Arguments

`builder`: builder for user cut.

CallbackBase.AddUserCuts()

Add user cuts to model.

Synopsis

```
void AddUserCuts(ConstrBuilderArray builders)
```

Arguments

`builders`: array of builders for user cuts.

CallbackBase.callback()

Pure virtual function defined in ICallback interface. User must implement it.

Synopsis

```
void callback()
```

CallbackBase.GetDblInfo()

Get double value of given information name in callback.

Synopsis

```
double GetDblInfo(string cbinfo)
```

Arguments

`cbinfo`: name of callback info.

Return

value of desired information.

CallbackBase.GetIncumbent()

Get best feasible solution of given variable in callback.

Synopsis

```
double GetIncumbent(Var var)
```

Arguments

`var`: given variable.

Return

best feasible solution of given variable.

CallbackBase.GetIncumbent()

Get best feasible solution of variables in callback.

Synopsis

```
double[] GetIncumbent(VarArray vars)
```

Arguments

`vars`: an array of variables.

Return

best feasible solution of desired variables.

CallbackBase.GetIncumbent()

Get best feasible solution of variables in callback.

Synopsis

```
double[] GetIncumbent(Var[] vars)
```

Arguments

`vars`: an array of variables.

Return

best feasible solution of desired variables.

CallbackBase.GetIncumbent()

Get best feasible solution of all variables in callback.

Synopsis

```
double[] GetIncumbent()
```

Return

best feasible solution of all variables.

CallbackBase.GetIntInfo()

Get integer value of given information name in callback.

Synopsis

```
int GetIntInfo(string cbinfo)
```

Arguments

`cbinfo`: name of callback info.

Return

value of desired information.

CallbackBase.GetRelaxSol()

Get LP-relaxation solution of given variable in callback.

Synopsis

```
double GetRelaxSol(Var var)
```

Arguments

`var`: given variable.

Return

LP-relaxation solution of given variable.

CallbackBase.GetRelaxSol()

Get LP-relaxation solution of variables in callback.

Synopsis

```
double[] GetRelaxSol(VarArray vars)
```

Arguments

`vars`: an array of variables.

Return

LP-relaxation solution of variables.

CallbackBase.GetRelaxSol()

Get LP-relaxation solution of variables in callback.

Synopsis

```
double[] GetRelaxSol(Var[] vars)
```

Arguments

`vars`: an array of variables.

Return

LP-relaxation solution of variables.

CallbackBase.GetRelaxSol()

Get LP-relaxation solution of all variables in callback.

Synopsis

```
double[] GetRelaxSol()
```

Return

LP-relaxation solution of all variables.

CallbackBase.GetSolution()

Get solution of given variable in callback.

Synopsis

```
double GetSolution(Var var)
```

Arguments

`var`: given variable.

Return

solution of given variable.

CallbackBase.GetSolution()

Get solution of variables in callback.

Synopsis

```
double[] GetSolution(VarArray vars)
```

Arguments

`vars`: an array of variables.

Return

solution of variables.

CallbackBase.GetSolution()

Get solution of variables in callback.

Synopsis

```
double[] GetSolution(Var[] vars)
```

Arguments

`vars`: an array of variables.

Return

solution of variables.

CallbackBase.GetSolution()

Get solution of all variables in callback.

Synopsis

```
double[] GetSolution()
```

Return

solution of all variables.

CallbackBase.Interrupt()

Interrupt solving problems in callback

Synopsis

```
void Interrupt()
```

CallbackBase.LoadSolution()

Load customized solution to model.

Synopsis

```
double LoadSolution()
```

Return

objective value of given solution.

CallbackBase.SetSolution()

Set solution of a given variable in callback.

Synopsis

```
void SetSolution(Var var, double val)
```

Arguments

`var`: a variable object.

`val`: double value.

CallbackBase.SetSolution()

Set solution of variables in callback.

Synopsis

```
void SetSolution(VarArray vars, double[] vals)
```

Arguments

`vars`: an array of variable objects.

`vals`: an array of double values.

CallbackBase.SetSolution()

Set solution of variables in callback.

Synopsis

```
void SetSolution(Var[] vars, double[] vals)
```

Arguments

`vars`: an array of variable objects.

`vals`: an array of double values.

CallbackBase.Where()

Get context in callback.

Synopsis

```
int Where()
```

Return

integer value of context.

19.2.41 ProbBuffer

Buffer object for COPT problem. ProbBuffer object holds the (MPS) problem in string format.

ProbBuffer.ProbBuffer()

Constructor of ProbBuffer object.

Synopsis

```
ProbBuffer(int sz)
```

Arguments

`sz`: initial size of the problem buffer.

ProbBuffer.GetData()

Get string of problem in problem buffer.

Synopsis

```
string GetData()
```

Return

string of problem in problem buffer.

ProbBuffer.Resize()

Resize buffer to given size, and zero-ended

Synopsis

```
void Resize(int sz)
```

Arguments

`sz`: new buffer size.

ProbBuffer.Size()

Get the size of problem buffer.

Synopsis

```
int Size()
```

Return

size of problem buffer.

19.2.42 CoptException

Copt exception object.

CoptException.CoptException()

Constructor of coptexception.

Synopsis

```
CoptException(int code, string msg)
```

Arguments

`code`: error code for exception.

`msg`: error message for exception.

CoptException.GetCode()

Get the error code associated with the exception.

Synopsis

```
int GetCode()
```

Return

the error code.

Chapter 20

Java API Reference

The **Cardinal Optimizer** provides a Java API library. This chapter documents all COPT Java constants and API functions for Java applications.

20.1 Constants

There are four types of constants defined in **Cardinal Optimizer**. They are general constants, information constants, attributes constants and parameters constants.

20.1.1 General Constants

For the contents of Java general constants, see *General Constants*.

General constants are defined in `Consts` class. User may refer general constants with namespace, that is, `copt.Consts.XXXX`.

20.1.2 Attributes

For the contents of Java attribute constants, see *Attributes*.

All COPT Java attributes are defined in `DblAttr` and `IntAttr` classes. User may refer double attributes by `copt.DblAttr.XXXX`, and integer attributes by `copt.IntAttr.XXXX`.

In the Java API, user can get the attribute value by specifying the attribute name. The two functions of obtaining attribute values are as follows, please refer to *Java API: Model Class* for details.

- `Model.getIntAttr()`: Get value of a COPT integer attribute.
- `Model.getDblAttr()`: Get value of a COPT double attribute.

20.1.3 Information

For the content of Java information constants, see *Information*.

In the Java API, information constants are defined in the `DblInfo` class. Users can access information constants through the prefix `copt` in the namespace (usually can be omitted) `copt.DblInfo.`

For instance, `copt.DblInfo.Obj` is the coefficients of variables in the objective function.

20.1.4 Callback Information

For the content of Java API callback information class constants, see *Callback Information*.

In the Java API, callback-related information constants are defined in the `CbInfo` class. Users can access information constants through the prefix `copt` in the namespace (usually can be omitted) `copt.CbInfo.`

For instance, `copt.CbInfo.BestObj` is the current best objective.

20.1.5 Parameters

For the contents of Java parameters constants, see *Parameters*.

All COPT Java parameters are defined in DblParam and IntParam classes. User may refer double parameters by `copt.DblParam.XXXX`, and integer parameters by `copt.IntParam.XXXX`.

In the Java API, user can get and set the parameter value by specifying the parameter name. The provided functions are as follows, please refer to *Java API: Model Class* for details.

- Get detailed information of the specified parameter (current value/max/min): `Model.getParamInfo()`
- Get the current value of the specified integer/double parameter: `Model.getIntParam()` / `Model.getDblParam()`
- Set the specified integer/double parameter value: `Model.setIntParam()` / `Model.setDblParam()`

20.2 Java Modeling Classes

This chapter documents COPT Java interface. Users may refer to Java classes described below for details of how to construct and solve Java models.

20.2.1 Envr

Essentially, any Java application using Cardinal Optimizer should start with a COPT environment. COPT models are always associated with a COPT environment. User must create an environment object before populating models. User generally only need a single environment object in program.

Envr.Envr()

Constructor of COPT Envr object.

Synopsis

```
Envr()
```

Envr.Envr()

Constructor of COPT Envr object, given a license folder.

Synopsis

```
Envr(String licDir)
```

Arguments

`licDir`: directory having local license or client config file.

Envr.Envr()

Constructor of COPT Envr object, given an Envr config object.

Synopsis

```
Envr(EnvrConfig config)
```

Arguments

`config`: Envr config object holding settings for remote connection.

Envr.close()

close remote connection and token becomes invalid for all problems in current envr.

Synopsis

```
void close()
```

Envr.createModel()

Create a model object.

Synopsis

```
Model createModel(String name)
```

Arguments

`name`: customized model name.

Return

a model object.

20.2.2 EnvrConfig

If user connects to COPT remote services, such as floating token server or compute cluster, it is necessary to add config settings with EnvrConfig object.

EnvrConfig.EnvrConfig()

Constructor of envr config object.

Synopsis

```
EnvrConfig()
```

EnvrConfig.set()

Set config settings in terms of name-value pair.

Synopsis

```
void set(String name, String value)
```

Arguments

`name`: keyword of a config setting.

`value`: value of a config setting.

20.2.3 Model

In general, a COPT model consists of a set of variables, a (linear) objective function on these variables, a set of constraints on there varaibles, etc. COPT model class encapsulates all required methods for constructing a COPT model.

Model.Model()

Constructor of model.

Synopsis

```
Model(Envr env, String name)
```

Arguments

`env`: associated environment object.

`name`: string of model name.

Model.addCone()

Add a cone constraint to model.

Synopsis

```
Cone addCone(
    int dim,
    int type,
    char[] pvtype,
    String prefix)
```

Arguments

`dim`: dimension of the cone constraint.

`type`: type of a cone constraint.

`pvtype`: types of variables in the cone.

`prefix`: name prefix of variables in the cone.

Return

new cone constraint object.

Model.addCone()

Add a cone constraint to model.

Synopsis

```
Cone addCone(ConeBuilder builder)
```

Arguments

`builder`: builder for new cone constraint.

Return

new cone constraint object.

Model.addCone()

Add a cone constraint to model.

Synopsis

```
Cone addCone(Var[] vars, int type)
```

Arguments

`vars`: variables that participate in the cone constraint.

`type`: type of a cone constraint.

Return

new cone constraint object.

Model.addCone()

Add a cone constraint to model.

Synopsis

```
Cone addCone(VarArray vars, int type)
```

Arguments

`vars`: variables that participate in the cone constraint.

`type`: type of a cone constraint.

Return

new cone constraint object.

Model.addConstr()

Add a linear constraint to model.

Synopsis

```
Constraint addConstr(
    Expr expr,
    char sense,
    double rhs,
    String name)
```

Arguments

`expr`: expression for the new contraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side value for the new constraint.

`name`: name of new constraint.

Return

new constraint object.

Model.addConstr()

Add a linear constraint to model.

Synopsis

```
Constraint addConstr(
    Expr expr,
    char sense,
    Var var,
    String name)
```

Arguments

`expr`: expression for the new contraint.

`sense`: sense for new linear constraint, other than range sense.

`var`: varible for the new constraint.

`name`: name of new constraint.

Return

new constraint object.

Model.addConstr()

Add a linear constraint to model.

Synopsis

```
Constraint addConstr(
    Expr lhs,
    char sense,
    Expr rhs,
    String name)
```

Arguments

`lhs`: left hand side expression for the new constraint.

`sense`: sense for new linear constraint, other than range sense.

`rhs`: right hand side expression for the new constraint.

`name`: name of new constraint.

Return

new constraint object.

Model.addConstr()

Add a linear constraint to model.

Synopsis

```
Constraint addConstr(
    Expr expr,
    double lb,
    double ub,
    String name)
```

Arguments

`expr`: expression for the new constraint.

`lb`: lower bound for the new constraint.

`ub`: upper bound for the new constraint

`name`: name of new constraint.

Return

new constraint object.

Model.addConstr()

Add a linear constraint to a model.

Synopsis

```
Constraint addConstr(ConstrBuilder builder, String name)
```

Arguments

`builder`: builder for the new constraint.

`name`: name of new constraint.

Return

new constraint object.

Model.addConstrs()

Add linear constraints to model.

Synopsis

```
ConstrArray addConstrs(
    int count,
    char[] senses,
    double[] rhss,
    String prefix)
```

Arguments

`count`: number of constraints added to model.

`senses`: sense array for new linear constraints, other than range sense.

`rhss`: right hand side values for new variables.

`prefix`: name prefix for new constraints.

Return

array of new constraint objects.

Model.addConstrs()

Add linear constraints to a model.

Synopsis

```
ConstrArray addConstrs(
    int count,
    double[] lbs,
    double[] ubs,
    String prefix)
```

Arguments

`count`: number of constraints added to the model.

`lbs`: lower bounds of new constraints.

`ubs`: upper bounds of new constraints.

`prefix`: name prefix for new constraints.

Return

array of new constraint objects.

Model.addConstrs()

Add linear constraints to a model.

Synopsis

```
ConstrArray addConstrs(ConstrBuilderArray builders, String prefix)
```

Arguments

`builders`: builders for new constraints.

`prefix`: name prefix for new constraints.

Return

array of new constraint objects.

Model.addDenseMat()

Add a dense symmetric matrix to a model.

Synopsis

```
SymMatrix addDenseMat(int dim, double[] vals)
```

Arguments

`dim`: dimension of the dense symmetric matrix.

`vals`: array of non-zero elements, filled in column-wise up to len or max length of symmetric matrix.

Return

new symmetric matrix object.

Model.addDenseMat()

Add a dense symmetric matrix to a model.

Synopsis

```
SymMatrix addDenseMat(int dim, double val)
```

Arguments

`dim`: dimension of dense symmetric matrix.

`val`: value to fill dense symmetric matrix.

Return

new symmetric matrix object.

Model.addDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix addDiagMat(int dim, double val)
```

Arguments

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

Return

new diagonal matrix object.

Model.addDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix addDiagMat(int dim, double[] vals)
```

Arguments

`dim`: dimension of diagonal matrix.

`vals`: array of values of diagonal elements.

Return

new diagonal matrix object.

Model.addDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix addDiagMat(
    int dim,
    double val,
    int offset)
```

Arguments

`dim`: dimension of diagonal matrix.

`val`: value to fill diagonal elements.

`offset`: shift distance against diagonal line.

Return

new diagonal matrix object.

Model.addDiagMat()

Add a diagonal matrix to a model.

Synopsis

```
SymMatrix addDiagMat(
    int dim,
    double[] vals,
    int offset)
```

Arguments

`dim`: dimension of diagonal matrix.

`vals`: array of values of diagonal elements.

`offset`: shift distance against diagonal line.

Return

new diagonal matrix object.

Model.addEyeMat()

Add an identity matrix to a model.

Synopsis

```
SymMatrix addEyeMat(int dim)
```

Arguments

`dim`: dimension of identity matrix.

Return

new identity matrix object.

Model.addGenConstrIndicator()

Add a general constraint of type indicator to model.

Synopsis

```
GenConstr addGenConstrIndicator(GenConstrBuilder builder)
```

Arguments

`builder`: builder for the general constraint.

Return

new general constraint object of type indicator.

Model.addGenConstrIndicator()

Add a general constraint of type indicator to model.

Synopsis

```
GenConstr addGenConstrIndicator(
    Var binvar,
    int binval,
    ConstrBuilder builder)
```

Arguments

`binvar`: binary indicator variable.

`binval`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`builder`: builder for linear constraint.

Return

new general constraint object of type indicator.

Model.addGenConstrIndicator()

Add a general constraint of type indicator to model.

Synopsis

```
GenConstr addGenConstrIndicator(
    Var binvar,
    int binval,
    Expr expr,
    char sense,
    double rhs)
```

Arguments

`binvar`: binary indicator variable.

`binval`: value for binary indicator variable that force a linear constraint to be satisfied(0 or 1).

`expr`: expression for new linear contraint.

`sense`: sense for new linear constraint.

`rhs`: right hand side value for new linear constraint.

Return

new general constraint object of type indicator.

Model.addLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void addLazyConstr(
    Expr lhs,
    char sense,
    double rhs,
    String name)
```

Arguments

`lhs`: expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

`name`: optional, name of lazy constraint.

Model.addLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void addLazyConstr(
    Expr lhs,
    char sense,
    Expr rhs,
    String name)
```

Arguments

`lhs`: left hand side expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy contraint.

`name`: optional, name of lazy constraint.

Model.addLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void addLazyConstr(ConstrBuilder builder, String name)
```

Arguments

`builder`: builder for lazy contraint.

`name`: optional, name of lazy constraint.

Model.addLazyConstrs()

Add lazy constraints to model.

Synopsis

```
void addLazyConstrs(ConstrBuilderArray builders, String prefix)
```

Arguments

`builders`: array of builders for lazy contraints.

`prefix`: name prefix of new lazy constraints.

Model.addOnesMat()

Add a dense symmetric matrix of value one to a model.

Synopsis

```
SymMatrix addOnesMat(int dim)
```

Arguments

`dim`: dimension of dense symmetric matrix.

Return

new symmetric matrix object.

Model.addPsdConstr()

Add a PSD constraint to model.

Synopsis

```
PsdConstraint addPsdConstr(
    PsdExpr expr,
    char sense,
    double rhs,
    String name)
```

Arguments

`expr`: PSD expression for new PSD contraint.

`sense`: sense for new PSD constraint.

`rhs`: double value at right side of the new PSD constraint.

`name`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model.addPsdConstr()

Add a PSD constraint to model.

Synopsis

```
PsdConstraint addPsdConstr(
    PsdExpr expr,
    double lb,
    double ub,
    String name)
```

Arguments

`expr`: expression for new PSD constraint.

`lb`: lower bound for ew PSD constraint.

`ub`: upper bound for new PSD constraint

`name`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model.addPsdConstr()

Add a PSD constraint to model.

Synopsis

```
PsdConstraint addPsdConstr(
    PsdExpr lhs,
    char sense,
    PsdExpr rhs,
    String name)
```

Arguments

`lhs`: PSD expression at left side of new PSD constraint.

`sense`: sense for new PSD constraint.

`rhs`: PSD expression at right side of new PSD constraint.

`name`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model.addPsdConstr()

Add a PSD constraint to a model.

Synopsis

```
PsdConstraint addPsdConstr(PsdConstrBuilder builder, String name)
```

Arguments

`builder`: builder for new PSD constraint.

`name`: optional, name of new PSD constraint.

Return

new PSD constraint object.

Model.addPsdVar()

Add a new PSD variable to model.

Synopsis

```
PsdVar addPsdVar(int dim, String name)
```

Arguments

`dim`: dimension of new PSD variable.

`name`: name of new PSD variable.

Return

PSD variable object.

Model.addPsdVars()

Add new PSD variables to model.

Synopsis

```
PsdVarArray addPsdVars(
    int count,
    int[] dims,
    String prefix)
```

Arguments

`count`: number of new PSD variables.

`dims`: array of dimensions of new PSD variables.

`prefix`: name prefix of new PSD variables.

Return

array of PSD variable objects.

Model.addQConstr()

Add a quadratic constraint to model.

Synopsis

```
QConstraint addQConstr(
    QuadExpr expr,
    char sense,
    double rhs,
    String name)
```

Arguments

`expr`: quadratic expression for the new contraint.

`sense`: sense for new quadratic constraint.

`rhs`: double value at right side of the new quadratic constraint.

`name`: optional, name of new quadratic constraint.

Return

new quadratic constraint object.

Model.addQConstr()

Add a quadratic constraint to model.

Synopsis

```
QConstraint addQConstr(
    QuadExpr lhs,
    char sense,
    QuadExpr rhs,
    String name)
```

Arguments

`lhs`: quadratic expression at left side of new quadratic constraint.

`sense`: sense for new quadratic constraint.

`rhs`: quadratic expression at right side of new quadratic constraint.

`name`: optional, name of new quadratic constraint.

Return

new quadratic constraint object.

Model.addQConstr()

Add a quadratic constraint to a model.

Synopsis

```
QConstraint addQConstr(QConstrBuilder builder, String name)
```

Arguments

`builder`: builder for the new quadratic constraint.

`name`: optional, name of new quadratic constraint.

Return

new quadratic constraint object.

Model.addSos()

Add a SOS constraint to model.

Synopsis

```
Sos addSos(
    VarArray vars,
    double[] weights,
    int type)
```

Arguments

`vars`: variables that participate in the SOS constraint.

`weights`: weights for variables in the SOS constraint.

`type`: type of SOS constraint.

Return

new SOS constraint object.

Model.addSos()

Add a SOS constraint to model.

Synopsis

```
Sos addSos(SosBuilder builder)
```

Arguments

`builder`: builder for new SOS constraint.

Return

new SOS constraint object.

Model.addSos()

Add a SOS constraint to model.

Synopsis

```
Sos addSos(
    Var[] vars,
    double[] weights,
    int type)
```

Arguments

`vars`: variables that participate in the SOS constraint.

`weights`: weights for variables in the SOS constraint.

`type`: type of SOS constraint.

Return

new SOS constraint object.

Model.addSparseMat()

Add a sparse symmetric matrix to a model.

Synopsis

```
SymMatrix addSparseMat(
    int dim,
    int nElems,
    int[] rows,
    int[] cols,
    double[] vals)
```

Arguments

`dim`: dimension of the sparse symmetric matrix.

`nElems`: number of non-zero elements in the sparse symmetric matrix.

`rows`: array of row indexes of non-zero elements.

`cols`: array of col indexes of non-zero elements.

`vals`: array of values of non-zero elements.

Return

new symmetric matrix object.

Model.addSymMat()

Given a symmetric matrix expression, add results matrix to model.

Synopsis

```
SymMatrix addSymMat(SymMatExpr expr)
```

Arguments

`expr`: symmetric matrix expression object.

Return

results symmetric matrix object.

Model.addUserCut()

Add a user cut to model.

Synopsis

```
void addUserCut(
    Expr lhs,
    char sense,
    double rhs,
    String name)
```

Arguments

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

`name`: optional, name of user cut.

Model.addUserCut()

Add a user cut to model.

Synopsis

```
void addUserCut(
    Expr lhs,
    char sense,
    Expr rhs,
    String name)
```

Arguments

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

`name`: optional, name of user cut.

Model.addUserCut()

Add a user cut to model.

Synopsis

```
void addUserCut(ConstrBuilder builder, String name)
```

Arguments

`builder`: builder for user cut.

`name`: optional, name of user cut.

Model.addUserCuts()

Add user cuts to model.

Synopsis

```
void addUserCuts(ConstrBuilderArray builders, String prefix)
```

Arguments

`builders`: array of builders for user cuts.

`prefix`: name prefix of new user cuts.

Model.addVar()

Add a variable and the associated non-zero coefficients as column.

Synopsis

```
Var addVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    String name)
```

Arguments

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`name`: name for new variable.

Return

new variable object.

Model.addVar()

Add a variable and the associated non-zero coefficients as column.

Synopsis

```
Var addVar(
    double lb,
    double ub,
    double obj,
    char vtype,
    Column col,
    String name)
```

Arguments

`lb`: lower bound for new variable.

`ub`: upper bound for new variable.

`obj`: objective coefficient for new variable.

`vtype`: variable type for new variable.

`col`: column object for specifying a set of constraints to which the variable belongs.

`name`: name for new variable.

Return

new variable object.

Model.addVars()

Add new variables to model.

Synopsis

```
VarArray addVars(
    int count,
    char vtype,
    String prefix)
```

Arguments

`count`: the number of variables to add.

`vtype`: variable types for new variables.

`prefix`: prefix part for names of new variables.

Return

array of new variable objects.

Model.addVars()

Add new variables to model.

Synopsis

```
VarArray addVars(
    int count,
    double lb,
    double ub,
    double obj,
    char vtype,
    String prefix)
```

Arguments

`count`: the number of variables to add.

`lb`: lower bound for new variables.

`ub`: upper bound for new variables.

`obj`: objective coefficient for new variables.

`vtype`: variable type for new variables.

`prefix`: prefix part for names of new variables.

Return

array of new variable objects.

Model.addVars()

Add new variables to model.

Synopsis

```
VarArray addVars(
    int count,
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    String prefix)
```

Arguments

`count`: the number of variables to add.

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`prefix`: prefix part for names of new variables.

Return

array of new variable objects.

Model.addVars()

Add new variables to model.

Synopsis

```
VarArray addVars(
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    Column[] cols,
    String prefix)
```

Arguments

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`cols`: column objects for specifying a set of constraints to which each new variable belongs.

`prefix`: prefix part for names of new variables.

Return

array of new variable objects.

Model.addVars()

Add new variables to model.

Synopsis

```
VarArray addVars(
    double[] lbs,
    double[] ubs,
    double[] objs,
    char[] types,
    ColumnArray cols,
    String prefix)
```

Arguments

`lbs`: lower bounds for new variables. if NULL, lower bounds are 0.0.

`ubs`: upper bounds for new variables. if NULL, upper bounds are infinity or 1 for binary variables.

`objs`: objective coefficients for new variables. if NULL, objective coefficients are 0.0.

`types`: variable types for new variables. if NULL, variable types are continuous.

`cols`: columnarray for specifying a set of constraints to which each new variable belongs.

`prefix`: prefix part for names of new variables.

Return

array of new variable objects.

Model.clear()

Clear all settings including problem itself.

Synopsis

```
void clear()
```

Model.clone()

Deep copy a new model object.

Synopsis

```
Model clone()
```

Return

cloned model object.

Model.computeIIS()

Compute IIS for infeasible model.

Synopsis

```
void computeIIS()
```

Model.delPsdObj()

delete PSD part of objective in model.

Synopsis

```
void delPsdObj()
```

Model.delQuadObj()

delete quadratic part of objective in model.

Synopsis

```
void delQuadObj()
```

Model.feasRelax()

Compute feasibility relaxation for infeasible model.

Synopsis

```
void feasRelax(
    VarArray vars,
    double[] colLowPen,
    double[] colUppPen,
    ConstrArray cons,
    double[] rowBndPen,
    double[] rowUppPen)
```

Arguments

`vars`: an array of variables.

`colLowPen`: penalties for lower bounds of variables.

`colUppPen`: penalties for upper bounds of variables.

`cons`: an array of constraints.

`rowBndPen`: penalties for right hand sides of constraints.

`rowUppPen`: penalties for upper bounds of range constraints.

Model.feasRelax()

Compute feasibility relaxation for infeasible model.

Synopsis

```
void feasRelax(int ifRelaxVars, int ifRelaxCons)
```

Arguments

`ifRelaxVars`: whether to relax variables.

`ifRelaxCons`: whether to relax constraints.

Model.get()

Query values of double parameter or double attribute, associated with variables.

Synopsis

```
double[] get(String name, Var[] vars)
```

Arguments

`name`: name of double parameter or double attribute.

`vars`: a list of interested variables.

Return

values of parameter or attrribute.

Model.get()

Query values of double parameter or double attribute, associated with variables.

Synopsis

```
double[] get(String name, VarArray vars)
```

Arguments

`name`: name of double parameter or double attribute.

`vars`: array of interested variables.

Return

values of parameter or attrribute.

Model.get()

Query values of parameter or attribute, associated with constraints.

Synopsis

```
double[] get(String name, Constraint[] constrs)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: a list of interested constraints.

Return

values of parameter or attrribute.

Model.get()

Query values of parameter or attribute, associated with constraints.

Synopsis

```
double[] get(String name, ConstrArray constrs)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: array of interested constraints.

Return

values of parameter or attrribute.

Model.get()

Query values of parameter or attribute, associated with quadratic constraints.

Synopsis

```
double[] get(String name, QConstraint[] constrs)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: a list of interested quadratic constraints.

Return

values of parameter or attrribute.

Model.get()

Query values of parameter or attribute, associated with quadratic constraints.

Synopsis

```
double[] get(String name, QConstrArray constrs)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: array of interested quadratic constraints.

Return

values of parameter or attrribute.

Model.get()

Query values of parameter or attribute, associated with PSD constraints.

Synopsis

```
double[] get(String name, PsdConstraint[] constrs)
```

Arguments

`name`: name of double parameter or attribute.

`constrs`: a list of desired PSD constraints.

Return

output array of parameter or attribute values.

Model.get()

Query values of parameter or attribute, associated with PSD constraints.

Synopsis

```
double[] get(String name, PsdConstrArray constrs)
```

Arguments

`name`: name of double parameter or attribute.

`constrs`: a list of desired PSD constraints.

Return

output array of parameter or attribute values.

Model.getCoeff()

Get the coefficient of variable in linear constraint.

Synopsis

```
double getCoeff(Constraint constr, Var var)
```

Arguments

`constr`: The requested constraint.

`var`: The requested variable.

Return

The requested coefficient.

Model.getCol()

Get a column object that have a list of constraints in which the variable participates.

Synopsis

```
Column getCol(Var var)
```

Arguments

`var`: a variable object.

Return

a column object associated with a variable.

Model.getColBasis()

Get status of column basis.

Synopsis

```
int[] getColBasis()
```

Return

basis status.

Model.getCone()

Get a cone constraint of given index in model.

Synopsis

```
Cone getCone(int idx)
```

Arguments

`idx`: index of the desired cone constraint.

Return

the desired cone constraint object.

Model.getConeBuilders()

Get builders of all cone constraints in model.

Synopsis

```
ConeBuilderArray getConeBuilders()
```

Return

array object of cone constraint builders.

Model.getConeBuilders()

Get builders of given cone constraints in model.

Synopsis

```
ConeBuilderArray getConeBuilders(Cone[] cones)
```

Arguments

`cones`: array of cone constraints.

Return

array object of desired cone constraint builders.

Model.getConeBuilders()

Get builders of given cone constraints in model.

Synopsis

```
ConeBuilderArray getConeBuilders(ConeArray cones)
```

Arguments

`cones`: array of cone constraints.

Return

array object of desired cone constraint builders.

Model.getCones()

Get all cone constraints in model.

Synopsis

```
ConeArray getCones()
```

Return

array object of cone constraints.

Model.getConstr()

Get a constraint of given index in model.

Synopsis

```
Constraint getConstr(int idx)
```

Arguments

`idx`: index of the desired constraint.

Return

the desired constraint object.

Model.getConstrBuilder()

Get builder of a constraint in model, including variables and associated coefficients, sense and RHS.

Synopsis

```
ConstrBuilder getConstrBuilder(Constraint constr)
```

Arguments

`constr`: a constraint object.

Return

constraint builder object.

Model.getConstrBuilders()

Get builders of all constraints in model.

Synopsis

```
ConstrBuilderArray getConstrBuilders()
```

Return

array object of constraint builders.

Model.getConstrByName()

Get a constraint of given name in model.

Synopsis

```
Constraint getConstrByName(String name)
```

Arguments

`name`: name of the desired constraint.

Return

the desired constraint object.

Model.getConstrLowerIIS()

Get IIS status of lower bounds of constraints.

Synopsis

```
int[] getConstrLowerIIS(ConstrArray constrs)
```

Arguments

`constrs`: Array of constraints.

Return

IIS status of lower bounds of constraints.

Model.getConstrLowerIIS()

Get IIS status of lower bounds of constraints.

Synopsis

```
int[] getConstrLowerIIS(Constraint[] constrs)
```

Arguments

`constrs`: Array of constraints.

Return

IIS status of lower bounds of constraints.

Model.getConstrs()

Get all constraints in model.

Synopsis

```
ConstrArray getConstrs()
```

Return

array object of constraints.

Model.getConstrUpperIIS()

Get IIS status of upper bounds of constraints.

Synopsis

```
int[] getConstrUpperIIS(ConstrArray constrs)
```

Arguments

`constrs`: Array of constraints.

Return

IIS status of upper bounds of constraints.

Model.getConstrUpperIIS()

Get IIS status of upper bounds of constraints.

Synopsis

```
int[] getConstrUpperIIS(Constraint[] constrs)
```

Arguments

`constrs`: Array of constraints.

Return

IIS status of upper bounds of constraints.

Model.getDblAttr()

Get value of a COPT double attribute.

Synopsis

```
double getDblAttr(String attr)
```

Arguments

`attr`: name of double attribute.

Return

value of double attribute.

Model.getDblParam()

Get value of a COPT double parameter.

Synopsis

```
double getDblParam(String param)
```

Arguments

`param`: name of double parameter.

Return

value of double parameter.

Model.getDblParamInfo()

Get current, default, minimum, maximum of COPT double parameter.

Synopsis

```
double[] getDblParamInfo(String name)
```

Arguments

`name`: name of integer parameter.

Return

current, default, minimum, maximum of COPT double parameter.

Model.getGenConstrIndicator()

Get builder of given general constraint of type indicator.

Synopsis

```
GenConstrBuilder getGenConstrIndicator(GenConstr indicator)
```

Arguments

`indicator`: a general constraint of type indicator.

Return

builder object of general constraint of type indicator.

Model.getIndicatorIIS()

Get IIS status of indicator constraints.

Synopsis

```
int[] getIndicatorIIS(GenConstrArray genconstrs)
```

Arguments

`genconstrs`: Array of indicator constraints.

Return

IIS status of indicator constraints.

Model.getIndicatorIIS()

Get IIS status of indicator constraints.

Synopsis

```
int[] getIndicatorIIS(GenConstr[] genconstrs)
```

Arguments

`genconstrs`: Array of indicator constraints.

Return

IIS status of indicator constraints.

Model.getIntAttr()

Get value of a COPT integer attribute

Synopsis

```
int getIntAttr(String attr)
```

Arguments

`attr`: name of integer attribute.

Return

value of integer attribute.

Model.getIntParam()

Get value of a COPT integer parameter.

Synopsis

```
int getIntParam(String param)
```

Arguments

`param`: name of integer parameter.

Return

value of integer parameter.

Model.getIntParamInfo()

Get current, default, minimum, maximum of COPT integer parameter.

Synopsis

```
int[] getIntParamInfo(String name)
```

Arguments

`name`: name of integer parameter.

Return

current, default, minimum, maximum of COPT integer parameter.

Model.getLpSolution()

Get LP solution.

Synopsis

```
Object[] getLpSolution()
```

Return

solution, slack, dual and reduced values.

Model.getObjective()

Get linear expression of objective for model.

Synopsis

```
Expr getObjective()
```

Return

an linear expression object.

Model.getPoolObjVal()

Get the idx-th objective value in solution pool.

Synopsis

```
double getPoolObjVal(int idx)
```

Arguments

`idx`: Index of solution.

Return

The requested objective value.

Model.getPoolSolution()

Get the idx-th solution in solution pool.

Synopsis

```
double[] getPoolSolution(int idx, VarArray vars)
```

Arguments

`idx`: Index of solution.

`vars`: The requested variables.

Return

The requested solution.

Model.getPoolSolution()

Get the idx-th solution in solution pool.

Synopsis

```
double[] getPoolSolution(int idx, Var[] vars)
```

Arguments

`idx`: Index of solution.

`vars`: The requested variables.

Return

The requested solution.

Model.getPsdCoeff()

Get the symmetric matrix of PSD variable in a PSD constraint.

Synopsis

```
SymMatrix getPsdCoeff(PsdConstraint constr, PsdVar var)
```

Arguments

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

Return

The associated coefficient matrix.

Model.getPsdConstr()

Get a quadratic constraint of given index in model.

Synopsis

```
PsdConstraint getPsdConstr(int idx)
```

Arguments

`idx`: index of the desired quadratic constraint.

Return

the desired quadratic constraint object.

Model.getPsdConstrBuilder()

Get builder of a PSD constraint in model, including PSD variables, sense and associated symmtric matrix.

Synopsis

```
PsdConstrBuilder getPsdConstrBuilder(PsdConstraint constr)
```

Arguments

`constr`: a PSD constraint object.

Return

PSD constraint builder object.

Model.getPsdConstrBuilders()

Get builders of all PSD constraints in model.

Synopsis

```
PsdConstrBuilderArray getPsdConstrBuilders()
```

Return

array object of PSD constraint builders.

Model.getPsdConstrByName()

Get a quadratic constraint of given name in model.

Synopsis

```
PsdConstraint getPsdConstrByName(String name)
```

Arguments

`name`: name of the desired constraint.

Return

the desired quadratic constraint object.

Model.getPsdConstrs()

Get all PSD constraints in model.

Synopsis

```
PsdConstrArray getPsdConstrs()
```

Return

array object of PSD constraints.

Model.getPsdObjective()

Get PSD objective of model.

Synopsis

```
PsdExpr getPsdObjective()
```

Return

a PSD expression object.

Model.getPsdRow()

Get PSD variables and associated symmetric matrix that participate in a PSD constraint.

Synopsis

```
PsdExpr getPsdRow(PsdConstraint constr)
```

Arguments

`constr`: a PSD constraint object.

Return

PSD expression object of the PSD constraint.

Model.getPsdVar()

Get a PSD variable of given index in model.

Synopsis

```
PsdVar getPsdVar(int idx)
```

Arguments

`idx`: index of the desired PSD variable.

Return

the desired PSD variable object.

Model.getPsdVarByName()

Get a PSD variable of given name in model.

Synopsis

```
PsdVar getPsdVarByName(String name)
```

Arguments

`name`: name of the desired PSD variable.

Return

the desired PSD variable object.

Model.getPsdVars()

Get all PSD variables in model.

Synopsis

```
PsdVarArray getPsdVars()
```

Return

array object of PSD variables.

Model.getQConstr()

Get a quadratic constraint of given index in model.

Synopsis

```
QConstraint getQConstr(int idx)
```

Arguments

`idx`: index of the desired quadratic constraint.

Return

the desired quadratic constraint object.

Model.getQConstrBuilder()

Get builder of a quadratic constraint in model, including variables and associated coefficients, sense and RHS.

Synopsis

```
QConstrBuilder getQConstrBuilder(QConstraint constr)
```

Arguments

`constr`: a constraint object.

Return

constraint builder object.

Model.getQConstrBuilders()

Get builders of all constraints in model.

Synopsis

```
QConstrBuilderArray getQConstrBuilders()
```

Return

array object of constraint builders.

Model.getQConstrByName()

Get a quadratic constraint of given name in model.

Synopsis

```
QConstraint getQConstrByName(String name)
```

Arguments

`name`: name of the desired constraint.

Return

the desired quadratic constraint object.

Model.getQConstrs()

Get all quadratic constraints in model.

Synopsis

```
QConstrArray getQConstrs()
```

Return

array object of quadratic constraints.

Model.getQuadObjective()

Get quadratic objective of model.

Synopsis

```
QuadExpr getQuadObjective()
```

Return

a quadratic expression object.

Model.getQuadRow()

Get quadratic expression that participate in quadratic constraint.

Synopsis

```
QuadExpr getQuadRow(QConstraint constr)
```

Arguments

`constr`: a quadratic constraint object.

Return

quadratic expression object of the constraint.

Model.getRow()

Get variables that participate in a constraint, and the associated coefficients.

Synopsis

```
Expr getRow(Constraint constr)
```

Arguments

`constr`: a constraint object.

Return

expression object of the constraint.

Model.getRowBasis()

Get status of row basis.

Synopsis

```
int[] getRowBasis()
```

Return

basis status.

Model.getSolution()

Get MIP solution.

Synopsis

```
double[] getSolution()
```

Return

solution values.

Model.getSos()

Get a SOS constraint of given index in model.

Synopsis

```
Sos getSos(int idx)
```

Arguments

`idx`: index of the desired SOS constraint.

Return

the desired SOS constraint object.

Model.getSosBuilders()

Get builders of all SOS constraints in model.

Synopsis

```
SosBuilderArray getSosBuilders()
```

Return

array object of SOS constraint builders.

Model.getSosBuilders()

Get builders of given SOS constraints in model.

Synopsis

```
SosBuilderArray getSosBuilders(Sos[] soss)
```

Arguments

`soss`: array of SOS constraints.

Return

array object of desired SOS constraint builders.

Model.getSosBuilders()

Get builders of given SOS constraints in model.

Synopsis

```
SosBuilderArray getSosBuilders(SosArray soss)
```

Arguments

`soss`: array of SOS constraints.

Return

array object of desired SOS constraint builders.

Model.getSOSIIS()

Get IIS status of SOS constraints.

Synopsis

```
int[] getSOSIIS(SosArray soss)
```

Arguments

`soss`: Array of SOS constraints.

Return

IIS status of SOS constraints.

Model.getSOSIIS()

Get IIS status of SOS constraints.

Synopsis

```
int[] getSOSIIS(Sos[] soss)
```

Arguments

`soss`: Array of SOS constraints.

Return

IIS status of SOS constraints.

Model.getSoss()

Get all SOS constraints in model.

Synopsis

```
SosArray getSoss()
```

Return

array object of SOS constraints.

Model.getSymMat()

Get a symmetric matrix of given index in model.

Synopsis

```
SymMatrix getSymMat(int idx)
```

Arguments

`idx`: index of the desired symmetric matrix.

Return

the desired symmetric matrix object.

Model.getVar()

Get a variable of given index in model.

Synopsis

```
Var getVar(int idx)
```

Arguments

`idx`: index of the desired variable.

Return

the desired variable object.

Model.getVarByName()

Get a variable of given name in model.

Synopsis

```
Var getVarByName(String name)
```

Arguments

`name`: name of the desired variable.

Return

the desired variable object.

Model.getVarLowerIIS()

Get IIS status of lower bounds of variables.

Synopsis

```
int[] getVarLowerIIS(VarArray vars)
```

Arguments

`vars`: Array of variables.

Return

IIS status of lower bounds of variables.

Model.getVarLowerIIS()

Get IIS status of lower bounds of variables.

Synopsis

```
int[] getVarLowerIIS(Var[] vars)
```

Arguments

`vars`: Array of variables.

Return

IIS status of lower bounds of variables.

Model.getVars()

Get all variables in model.

Synopsis

```
VarArray getVars()
```

Return

variable array object.

Model.getVarUpperIIS()

Get IIS status of upper bounds of variables.

Synopsis

```
int[] getVarUpperIIS(VarArray vars)
```

Arguments

`vars`: Array of variables.

Return

IIS status of upper bounds of variables.

Model.getVarUpperIIS()

Get IIS status of upper bounds of variables.

Synopsis

```
int[] getVarUpperIIS(Var[] vars)
```

Arguments

`vars`: Array of variables.

Return

IIS status of upper bounds of variables.

Model.interrupt()

Interrupt optimization of current problem.

Synopsis

```
void interrupt()
```

Model.loadMipStart()

Load final initial values of variables to the problem.

Synopsis

```
void loadMipStart()
```

Model.loadTuneParam()

Load specified tuned parameters into model.

Synopsis

```
void loadTuneParam(int idx)
```

Arguments

`idx`: Index of tuned parameters.

Model.read()

Read problem, solution, basis, MIP start or COPT parameters from file.

Synopsis

```
void read(String filename)
```

Arguments

`filename`: an input file name.

Model.readBasis()

Read basis from file.

Synopsis

```
void readBasis(String filename)
```

Arguments

`filename`: an input file name.

Model.readBin()

Read problem in COPT binary format from file.

Synopsis

```
void readBin(String filename)
```

Arguments

`filename`: an input file name.

Model.readCbf()

Read problem in CBF format from file.

Synopsis

```
void readCbf(String filename)
```

Arguments

`filename`: an input file name.

Model.readLp()

Read problem in LP format from file.

Synopsis

```
void readLp(String filename)
```

Arguments

`filename`: an input file name.

Model.readMps()

Read problem in MPS format from file.

Synopsis

```
void readMps(String filename)
```

Arguments

`filename`: an input file name.

Model.readMst()

Read MIP start information from file.

Synopsis

```
void readMst(String filename)
```

Arguments

`filename`: an input file name.

Model.readParam()

Read COPT parameters from file.

Synopsis

```
void readParam(String filename)
```

Arguments

`filename`: an input file name.

Model.readSdpa()

Read problem in SDPA format from file.

Synopsis

```
void readSdpa(String filename)
```

Arguments

`filename`: an input file name.

Model.readSol()

Read solution from file.

Synopsis

```
void readSol(String filename)
```

Arguments

`filename`: an input file name.

Model.readTune()

Read tuning parameters from file.

Synopsis

```
void readTune(String filename)
```

Arguments

`filename`: an input file name.

Model.remove()

Remove an array of variables from model.

Synopsis

```
void remove(Var[] vars)
```

Arguments

`vars`: a list of variables.

Model.remove()

Remove array of variables from model.

Synopsis

```
void remove(VarArray vars)
```

Arguments

`vars`: an array of variables.

Model.remove()

Remove a list of constraints from model.

Synopsis

```
void remove(Constraint[] constrs)
```

Arguments

`constrs`: a list of constraints.

Model.remove()

Remove a list of constraints from model.

Synopsis

```
void remove(ConstrArray constrs)
```

Arguments

`constrs`: an array of constraints.

Model.remove()

Remove a list of SOS constraints from model.

Synopsis

```
void remove(Sos[] soss)
```

Arguments

`soss`: a list of SOS constraints.

Model.remove()

Remove a list of SOS constraints from model.

Synopsis

```
void remove(SosArray soss)
```

Arguments

`soss`: an array of SOS constraints.

Model.remove()

Remove a list of Cone constraints from model.

Synopsis

```
void remove(Cone[] cones)
```

Arguments

`cones`: a list of Cone constraints.

Model.remove()

Remove a list of Cone constraints from model.

Synopsis

```
void remove(ConeArray cones)
```

Arguments

`cones`: an array of Cone constraints.

Model.remove()

Remove a list of gernal constraints from model.

Synopsis

```
void remove(GenConstr[] genConstrs)
```

Arguments

`genConstrs`: a list of general constraints.

Model.remove()

Remove a list of gernal constraints from model.

Synopsis

```
void remove(GenConstrArray genConstrs)
```

Arguments

`genConstrs`: an array of general constraints.

Model.remove()

Remove a list of quadratic constraints from model.

Synopsis

```
void remove(QConstraint[] qconstrs)
```

Arguments

`qconstrs`: an array of quadratic constraints.

Model.remove()

Remove a list of quadratic constraints from model.

Synopsis

```
void remove(QConstrArray qconstrs)
```

Arguments

`qconstrs`: an array of quadratic constraints.

Model.remove()

Remove a list of PSD variables from model.

Synopsis

```
void remove(PsdVar[] vars)
```

Arguments

`vars`: an array of PSD variables.

Model.remove()

Remove a list of PSD variables from model.

Synopsis

```
void remove(PsdVarArray vars)
```

Arguments

`vars`: an array of PSD variables.

Model.remove()

Remove a list of PSD constraints from model.

Synopsis

```
void remove(PsdConstraint[] constrs)
```

Arguments

`constrs`: an array of PSD constraints.

Model.remove()

Remove a list of PSD constraints from model.

Synopsis

```
void remove(PsdConstrArray constrs)
```

Arguments

`constrs`: an array of PSD constraints.

Model.reset()

Reset solution only.

Synopsis

```
void reset()
```

Model.resetAll()

Reset solution and additional information.

Synopsis

```
void resetAll()
```

Model.resetParam()

Reset parameters to default settings.

Synopsis

```
void resetParam()
```

Model.set()

Set values of double parameter or double attribute, associated with variables.

Synopsis

```
void set(
    String name,
    Var[] vars,
    double[] vals)
```

Arguments

`name`: name of double parameter or double attribute.

`vars`: a list of interested variables.

`vals`: values of parameter or attrribute.

Model.set()

Set values of double parameter or double attribute, associated with variables.

Synopsis

```
void set(
    String name,
    VarArray vars,
    double[] vals)
```

Arguments

`name`: name of double parameter or double attribute.

`vars`: array of interested variables.

`vals`: values of parameter or attrribute.

Model.set()

Set values of parameter or attribute, associated with constraints.

Synopsis

```
void set(
    String name,
    Constraint[] constrs,
    double[] vals)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: a list of interested constraints.

`vals`: values of parameter or attrribute.

Model.set()

Set values of parameter or attribute, associated with constraints.

Synopsis

```
void set(
    String name,
    ConstrArray constrs,
    double[] vals)
```

Arguments

`name`: name of double parameter or double attribute.

`constrs`: array of interested constraints.

`vals`: values of parameter or attrribute.

Model.set()

Set values of parameter, associated with PSD constraints.

Synopsis

```
void set(
    String name,
    PsdConstraint[] constrs,
    double[] vals)
```

Arguments

`name`: name of double parameter.

`constrs`: a list of desired PSD constraints.

`vals`: array of values of parameter.

Model.set()

Set values of parameter, associated with PSD constraints.

Synopsis

```
void set(
    String name,
    PsdConstrArray constrs,
    double[] vals)
```

Arguments

`name`: name of double parameter.

`constrs`: a list of desired PSD constraints.

`vals`: array of values of parameter.

Model.setBasis()

Set column and row basis status to model.

Synopsis

```
void setBasis(int[] colbasis, int[] rowbasis)
```

Arguments

`colbasis`: status of column basis.

`rowbasis`: status of row basis.

Model.setCallback()

Set user callback to COPT model.

Synopsis

```
void setCallback(CallbackBase cb, int cbctx)
```

Arguments

`cb`: user callback instance, inheriting from CallbackBase class.

`cbctx`: COPT callback context.

Model.setCoeff()

Set the coefficient of a variable in a linear constraint.

Synopsis

```
void setCoeff(
    Constraint constr,
    Var var,
    double newVal)
```

Arguments

`constr`: The requested constraint.

`var`: The requested variable.

`newVal`: New coefficient.

Model.setDblParam()

Set value of a COPT double parameter.

Synopsis

```
void setDblParam(String param, double val)
```

Arguments

`param`: name of double parameter.

`val`: double value.

Model.setIntParam()

Set value of a COPT integer parameter.

Synopsis

```
void setIntParam(String param, int val)
```

Arguments

`param`: name of integer parameter.

`val`: integer value.

Model.setLpSolution()

Set LP solution.

Synopsis

```
void setLpSolution(
    double[] value,
    double[] slack,
    double[] rowDual,
    double[] redCost)
```

Arguments

`value`: solution of variables.

`slack`: solution of slack variables.

`rowDual`: dual value of slack variables.

`redCost`: dual value of variables.

Model.setMipStart()

Set initial values for variables of given number, starting from the first one.

Synopsis

```
void setMipStart(int count, double[] vals)
```

Arguments

`count`: the number of variables to set.

`vals`: values of variables.

Model.setMipStart()

Set initial value for the specified variable.

Synopsis

```
void setMipStart(Var var, double val)
```

Arguments

`var`: an interested variable.

`val`: initial value of the variable.

Model.setMipStart()

Set initial value for the specified variable.

Synopsis

```
void setMipStart(Var[] vars, double[] vals)
```

Arguments

`vars`: a list of interested variables.

`vals`: initial values of the variables.

Model.setMipStart()

Set initial values for an array of variables.

Synopsis

```
void setMipStart(VarArray vars, double[] vals)
```

Arguments

`vars`: a list of interested variables.

`vals`: initial values of variables.

Model.setObjConst()

Set objective constant.

Synopsis

```
void setObjConst(double constant)
```

Arguments

`constant`: constant value to set.

Model.setObjective()

Set objective for model.

Synopsis

```
void setObjective(Expr expr, int sense)
```

Arguments

`expr`: expression of the objective.

`sense`: COPT_MINIMIZE for minimization and COPT_MAXIMIZE for maximization.

Model.setObjSense()

Set objective sense for model.

Synopsis

```
void setObjSense(int sense)
```

Arguments

`sense`: the objective sense.

Model.setPsdCoeff()

Set the coefficient matrix of a PSD variable in a PSD constraint.

Synopsis

```
void setPsdCoeff(
    PsdConstraint constr,
    PsdVar var,
    SymMatrix mat)
```

Arguments

`constr`: The desired PSD constraint.

`var`: The desired PSD variable.

`mat`: new coefficient matrix.

Model.setPsdObjective()

Set PSD objective for model.

Synopsis

```
void setPsdObjective(PsdExpr expr, int sense)
```

Arguments

`expr`: PSD expression of the objective.

`sense`: COPT sense.

Model.setQuadObjective()

Set quadratic objective for model.

Synopsis

```
void setQuadObjective(QuadExpr expr, int sense)
```

Arguments

`expr`: quadratic expression of the objective.

`sense`: default value 0 does not change COPT sense.

Model.setSlackBasis()

Set slack basis to model.

Synopsis

```
void setSlackBasis()
```

Model.setSolverLogFile()

Set log file for COPT.

Synopsis

```
void setSolverLogFile(String filename)
```

Arguments

`filename`: log file name.

Model.solve()

Solve the model as MIP.

Synopsis

```
void solve()
```

Model.solveLp()

Solve the model as LP.

Synopsis

```
void solveLp()
```

Model.tune()

Tune model.

Synopsis

```
void tune()
```

Model.write()

Output problem, solution, basis, MIP start or modified COPT parameters to file.

Synopsis

```
void write(String filename)
```

Arguments

`filename`: an output file name.

Model.writeBasis()

Output optimal basis to a file of type '.bas'.

Synopsis

```
void writeBasis(String filename)
```

Arguments

`filename`: an output file name.

Model.writeBin()

Output problem to a file as COPT binary format.

Synopsis

```
void writeBin(String filename)
```

Arguments

`filename`: an output file name.

Model.writeIIS()

Output IIS to file.

Synopsis

```
void writeIIS(String filename)
```

Arguments

`filename`: Output file name.

Model.writeLp()

Output problem to a file as LP format.

Synopsis

```
void writeLp(String filename)
```

Arguments

`filename`: an output file name.

Model.writeMps()

Output problem to a file as MPS format.

Synopsis

```
void writeMps(String filename)
```

Arguments

`filename`: an output file name.

Model.writeMpsStr()

Output MPS problem to problem buffer.

Synopsis

```
ProbBuffer writeMpsStr()
```

Return

problem buffer for string of MPS problem.

Model.writeMst()

Output MIP start information to a file of type '.mst'.

Synopsis

```
void writeMst(String filename)
```

Arguments

`filename`: an output file name.

Model.writeParam()

Output modified COPT parameters to a file of type '.par'.

Synopsis

```
void writeParam(String filename)
```

Arguments

`filename`: an output file name.

Model.writePoolSol()

Output selected pool solution to a file of type '.sol'.

Synopsis

```
void writePoolSol(int idx, String filename)
```

Arguments

`idx`: index of pool solution.

`filename`: an output file name.

Model.writeRelax()

Output feasibility relaxation problem to file.

Synopsis

```
void writeRelax(String filename)
```

Arguments

`filename`: Output file name.

Model.writeSol()

Output solution to a file of type '.sol'.

Synopsis

```
void writeSol(String filename)
```

Arguments

`filename`: an output file name.

Model.writeTuneParam()

Output specified tuned parameters to a file of type '.par'.

Synopsis

```
void writeTuneParam(int idx, String filename)
```

Arguments

`idx`: Index of tuned parameters.

`filename`: Output file name.

20.2.4 Var

COPT variable object. Variables are always associated with a particular model. User creates a variable object by adding a variable to a model, rather than by using constructor of Var class.

Var.get()

Get attribute value of the variable. Support "Value", "RedCost", "LB", "UB", and "Obj" attributes.

Synopsis

```
double get(String attr)
```

Arguments

`attr`: attribute name.

Return

attribute value.

Var.getBasis()

Get basis status of the variable.

Synopsis

```
int getBasis()
```

Return

Basis status.

Var.getIdx()

Get index of the variable.

Synopsis

```
int getIdx()
```

Return

variable index.

Var.getLowerIIS()

Get IIS status for lower bound of the variable.

Synopsis

```
int getLowerIIS()
```

Return

IIS status.

Var.getName()

Get name of the variable.

Synopsis

```
String getName()
```

Return

variable name.

Var.getType()

Get type of the variable.

Synopsis

```
char getType()
```

Return

variable type.

Var.getUpperIIS()

Get IIS status for upper bound of the variable.

Synopsis

```
int getUpperIIS()
```

Return

IIS status.

Var.remove()

Remove variable from model.

Synopsis

```
void remove()
```

Var.set()

Set attribute value of the variable. Support "LB", "UB" and "Obj" attributes.

Synopsis

```
void set(String attr, double val)
```

Arguments

`attr`: attribute name.

`val`: new value.

Var.setName()

Set name of the variable.

Synopsis

```
void setName(String name)
```

Arguments

`name`: variable name.

Var.setType()

Set type of the variable.

Synopsis

```
void setType(char vtype)
```

Arguments

`vtype`: variable type.

20.2.5 VarArray

COPT variable array object. To store and access a set of Java *Var* objects, Cardinal Optimizer provides Java VarArray class, which defines the following methods.

VarArray.VarArray()

Constructor of vararray.

Synopsis

```
VarArray()
```

VarArray.getVar()

Get idx-th variable object.

Synopsis

```
Var getVar(int idx)
```

Arguments

`idx`: index of the variable.

Return

variable object with index idx.

VarArray.pushBack()

Add a variable object to variable array.

Synopsis

```
void pushBack(Var var)
```

Arguments

`var`: a variable object.

VarArray.size()

Get the number of variable objects.

Synopsis

```
int size()
```

Return

number of variable objects.

20.2.6 Expr

COPT linear expression object. A linear expression consists of a constant term, a list of terms of variables and associated coefficients. Linear expressions are used to build constraints.

Expr.Expr()

Constructor of a constant linear expression with constant 0.0

Synopsis

```
Expr()
```

Expr.Expr()

Constructor of a constant linear expression.

Synopsis

```
Expr(double constant)
```

Arguments

`constant`: constant value in expression object.

Expr.Expr()

Constructor of a linear expression with one term.

Synopsis

```
Expr(Var var)
```

Arguments

`var`: variable for the added term.

Expr.Expr()

Constructor of a linear expression with one term.

Synopsis

```
Expr(Var var, double coeff)
```

Arguments

`var`: variable for the added term.

`coeff`: coefficent for the added term.

Expr.addConstant()

Add extra constant to the expression.

Synopsis

```
void addConstant(double constant)
```

Arguments

`constant`: delta value to be added to expression constant.

Expr.addExpr()

Add a linear expression to self.

Synopsis

```
void addExpr(Expr expr)
```

Arguments

`expr`: linear expression to be added.

Expr.addExpr()

Add a linear expression to self.

Synopsis

```
void addExpr(Expr expr, double mult)
```

Arguments

`expr`: linear expression to be added.

`mult`: multiplier constant.

Expr.addTerm()

Add a term to expression object.

Synopsis

```
void addTerm(Var var, double coeff)
```

Arguments

`var`: a variable for new term.

`coeff`: coefficient for new term.

Expr.addTerms()

Add terms to expression object.

Synopsis

```
void addTerms(Var[] vars, double coeff)
```

Arguments

`vars`: variables for added terms.

`coeff`: one coefficient for added terms.

Expr.addTerms()

Add terms to expression object.

Synopsis

```
void addTerms(Var[] vars, double[] coeffs)
```

Arguments

`vars`: variables for added terms.

`coeffs`: coefficients array for added terms.

Expr.addTerms()

Add terms to expression object.

Synopsis

```
void addTerms(VarArray vars, double coeff)
```

Arguments

`vars`: variables for added terms.

`coeff`: one coefficient for added terms.

Expr.addTerms()

Add terms to expression object.

Synopsis

```
void addTerms(VarArray vars, double[] coeffs)
```

Arguments

`vars`: variables for added terms.

`coeffs`: coefficients array for added terms.

Expr.clone()

Deep copy linear expression object.

Synopsis

```
Expr clone()
```

Return

cloned linear expression object.

Expr.evaluate()

evaluate linear expression after solving

Synopsis

```
double evaluate()
```

Return

value of linear expression

Expr.getCoeff()

Get coefficient from the i-th term in expression.

Synopsis

```
double getCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term in expression object.

Expr.getConstant()

Get constant in expression.

Synopsis

```
double getConstant()
```

Return

constant in expression.

Expr.getVar()

Get variable from the i-th term in expression.

Synopsis

```
Var getVar(int i)
```

Arguments

`i`: index of the term.

Return

variable of the i-th term in expression object.

Expr.remove()

Remove idx-th term from expression object.

Synopsis

```
void remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

Expr.remove()

Remove the term associated with variable from expression.

Synopsis

```
void remove(Var var)
```

Arguments

`var`: a variable whose term should be removed.

Expr.setCoeff()

Set coefficient for the i-th term in expression.

Synopsis

```
void setCoeff(int i, double val)
```

Arguments

`i`: index of the term.

`val`: coefficient of the term.

Expr.setConstant()

Set constant for the expression.

Synopsis

```
void setConstant(double constant)
```

Arguments

`constant`: the value of the constant.

Expr.size()

Get number of terms in expression.

Synopsis

```
long size()
```

Return

number of terms.

20.2.7 Constraint

COPT constraint object. Constraints are always associated with a particular model. User creates a constraint object by adding a constraint to a model, rather than by using constructor of Constraint class.

Constraint.get()

Get attribute value of the constraint. Support "Dual", "Slack", "LB", "UB" attributes.

Synopsis

```
double get(String attr)
```

Arguments

`attr`: name of the attribute being queried.

Return

attribute value.

Constraint.getBasis()

Get basis status of this constraint.

Synopsis

```
int getBasis()
```

Return

basis status.

Constraint.getIdx()

Get index of the constraint.

Synopsis

```
int getIdx()
```

Return

the index of the constraint.

Constraint.getLowerIIS()

Get IIS status for lower bound of the constraint.

Synopsis

```
int getLowerIIS()
```

Return

IIS status.

Constraint.getName()

Get name of the constraint.

Synopsis

```
String getName()
```

Return

the name of the constraint.

Constraint.getUpperIIS()

Get IIS status for upper bound of the constraint.

Synopsis

```
int getUpperIIS()
```

Return

IIS status.

Constraint.remove()

Remove this constraint from model.

Synopsis

```
void remove()
```

Constraint.set()

Set attribute value of the constraint. Support "LB" and "UB" attributes.

Synopsis

```
void set(String attr, double val)
```

Arguments

`attr`: name of the attribute.

`val`: new value.

Constraint.setName()

Set name for the constraint.

Synopsis

```
void setName(String name)
```

Arguments

`name`: the name to set.

20.2.8 ConstrArray

COPT constraint array object. To store and access a set of Java *Constraint* objects, Cardinal Optimizer provides Java ConstrArray class, which defines the following methods.

ConstrArray.ConstrArray()

Constructor of constrarray object.

Synopsis

```
ConstrArray()
```

ConstrArray.getConstr()

Get idx-th constraint object.

Synopsis

```
Constraint getConstr(int idx)
```

Arguments

`idx`: index of the constraint.

Return

constraint object with index idx.

ConstrArray.pushBack()

Add a constraint object to constraint array.

Synopsis

```
void pushBack(Constraint constr)
```

Arguments

`constr`: a constraint object.

ConstrArray.size()

Get the number of constraint objects.

Synopsis

```
int size()
```

Return

number of constraint objects.

20.2.9 ConstrBuilder

COPT constraint builder object. To help building a constraint, given a linear expression, constraint sense and right-hand side value, Cardinal Optimizer provides Java ConstrBuilder class, which defines the following methods.

ConstrBuilder.ConstrBuilder()

Constructor of constrbuilder object.

Synopsis

```
ConstrBuilder()
```

ConstrBuilder.getExpr()

Get expression associated with constraint.

Synopsis

```
Expr getExpr()
```

Return

expression object.

ConstrBuilder.getRange()

Get range from lower bound to upper bound of range constraint.

Synopsis

```
double getRange()
```

Return

length from lower bound to upper bound of the constraint.

ConstrBuilder.getSense()

Get sense associated with constraint.

Synopsis

```
char getSense()
```

Return

constraint sense.

ConstrBuilder.set()

Set detail of a constraint to its builder object.

Synopsis

```
void set(
    Expr expr,
    char sense,
    double rhs)
```

Arguments

`expr`: expression object at one side of the constraint

`sense`: constraint sense other than COPT_RANGE.

`rhs`: constant of right side of the constraint.

ConstrBuilder.setRange()

Set a range constraint to its builder.

Synopsis

```
void setRange(Expr expr, double range)
```

Arguments

`expr`: expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must greater than 0.

20.2.10 ConstrBuilderArray

COPT constraint builder array object. To store and access a set of Java *ConstrBuilder* objects, Cardinal Optimizer provides Java ConstrBuilderArray class, which defines the following methods.

ConstrBuilderArray.ConstrBuilderArray()

Constructor of constrbuilderarray object.

Synopsis

```
ConstrBuilderArray()
```

ConstrBuilderArray.getBuilder()

Get idx-th constraint builder object.

Synopsis

```
ConstrBuilder getBuilder(int idx)
```

Arguments

`idx`: index of the constraint builder.

Return

constraint builder object with index idx.

ConstrBuilderArray.pushBack()

Add a constraint builder object to constraint builder array.

Synopsis

```
void pushBack(ConstrBuilder builder)
```

Arguments

`builder`: a constraint builder object.

ConstrBuilderArray.size()

Get the number of constraint builder objects.

Synopsis

```
int size()
```

Return

number of constraint builder objects.

20.2.11 Column

COPT column object. A column consists of a list of constraints and associated coefficients. Columns are used to represent the set of constraints in which a variable participates, and the asssociated coefficents.

Column.Column()

Constructor of column.

Synopsis

```
Column()
```

Column.addColumn()

Add a column to self.

Synopsis

```
void addColumn(Column col)
```

Arguments

`col`: column object to be added.

Column.addColumn()

Add a column to self.

Synopsis

```
void addColumn(Column col, double mult)
```

Arguments

`col`: column object to be added.

`mult`: multiply constant.

Column.addTerm()

Add a term to column object.

Synopsis

```
void addTerm(Constraint constr, double coeff)
```

Arguments

`constr`: a constraint for new term.

`coeff`: coefficient for new term.

Column.addTerms()

Add terms to column object.

Synopsis

```
void addTerms(Constraint[] constrs, double coeff)
```

Arguments

`constrs`: constraints for added terms.

`coeff`: coefficient for added terms.

Column.addTerms()

Add terms to column object.

Synopsis

```
void addTerms(Constraint[] constrs, double[] coeffs)
```

Arguments

`constrs`: constraints for added terms.

`coeffs`: coefficients for added terms.

Column.addTerms()

Add terms to column object.

Synopsis

```
void addTerms(ConstrArray constrs, double coeff)
```

Arguments

`constrs`: constraints for added terms.

`coeff`: coefficient for added terms.

Column.addTerms()

Add terms to column object.

Synopsis

```
void addTerms(ConstrArray constrs, double[] coeffs)
```

Arguments

`constrs`: constraints for added terms.

`coeffs`: coefficients for added terms.

Column.clear()

Clear all terms.

Synopsis

```
void clear()
```

Column.clone()

Deep copy column object.

Synopsis

```
Column clone()
```

Return

cloned column object.

Column.getCoeff()

Get coefficient from the i-th term in column object.

Synopsis

```
double getCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term in column object.

Column.getConstr()

Get constraint from the i-th term in column object.

Synopsis

```
Constraint getConstr(int i)
```

Arguments

`i`: index of the term.

Return

constraint of the i-th term in column object.

Column.remove()

Remove i-th term from column object.

Synopsis

```
void remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

Column.remove()

Remove the term associated with constraint from column object.

Synopsis

```
void remove(Constraint constr)
```

Arguments

`constr`: a constraint whose term should be removed.

Column.size()

Get number of terms in column object.

Synopsis

```
int size()
```

Return

number of terms.

20.2.12 ColumnArray

COPT column array object. To store and access a set of Java *Column* objects, Cardinal Optimizer provides Java ColumnArray class, which defines the following methods.

ColumnArray.ColumnArray()

Constructor of columnarray object.

Synopsis

```
ColumnArray()
```

ColumnArray.clear()

Clear all column objects.

Synopsis

```
void clear()
```

ColumnArray.getColumn()

Get idx-th column object.

Synopsis

```
Column getColumn(int idx)
```

Arguments

`idx`: index of the column.

Return

column object with index idx.

ColumnArray.pushBack()

Add a column object to column array.

Synopsis

```
void pushBack(Column col)
```

Arguments

`col`: a column object.

ColumnArray.size()

Get the number of column objects.

Synopsis

```
int size()
```

Return

number of column objects.

20.2.13 Sos

COPT SOS constraint object. SOS constraints are always associated with a particular model. User creates an SOS constraint object by adding an SOS constraint to a model, rather than by using constructor of Sos class.

An SOS constraint can be type 1 or 2 (`COPT_SOS_TYPE1` or `COPT_SOS_TYPE2`).

Sos.getIdx()

Get the index of SOS constarint.

Synopsis

```
int getIdx()
```

Return

index of SOS constraint.

Sos.getIIS()

Get IIS status of the SOS constraint.

Synopsis

```
int getIIS()
```

Return

IIS status.

Sos.remove()

Remove the SOS constraint from model.

Synopsis

```
void remove()
```

20.2.14 SosArray

COPT SOS constraint array object. To store and access a set of Java *Sos* objects, Cardinal Optimizer provides Java SosArray class, which defines the following methods.

SosArray.SosArray()

Constructor of sosarray object.

Synopsis

```
SosArray()
```

SosArray.getSos()

Get idx-th SOS object.

Synopsis

```
Sos getSos(int idx)
```

Arguments

`idx`: index of SOS.

Return

SOS object with index idx.

SosArray.pushBack()

Add a SOS constraint object to SOS constraint array.

Synopsis

```
void pushBack(Sos sos)
```

Arguments

`sos`: a SOS constraint object.

SosArray.size()

Get the number of SOS constraint objects.

Synopsis

```
int size()
```

Return

number of SOS constraint objects.

20.2.15 SosBuilder

COPT SOS constraint builder object. To help building an SOS constraint, given the SOS type, a set of variables and associated weights, Cardinal Optimizer provides Java SosBuilder class, which defines the following methods.

SosBuilder.SosBuilder()

Constructor of sosbuilder object.

Synopsis

```
SosBuilder()
```

SosBuilder.getSize()

Get number of terms in SOS constraint.

Synopsis

```
int getSize()
```

Return

number of terms.

SosBuilder.getType()

Get type of SOS constraint.

Synopsis

```
int getType()
```

Return

type of SOS constraint.

SosBuilder.getVar()

Get variable from the idx-th term in SOS constraint.

Synopsis

```
Var getVar(int idx)
```

Arguments

`idx`: index of the term.

Return

variable of the idx-th term in SOS constraint.

SosBuilder.GetVars()

Get all variables in a SOS constraint.

Synopsis

```
VarArray GetVars()
```

Return

variables in a SOS constraint.

SosBuilder.getWeight()

Get weight from the idx-th term in SOS constraint.

Synopsis

```
double getWeight(int idx)
```

Arguments

`idx`: index of the term.

Return

weight of the idx-th term in SOS constraint.

SosBuilder.getWeights()

Get weights of all terms in SOS constraint.

Synopsis

```
double[] getWeights()
```

Return

array of weights.

SosBuilder.set()

Set variables and weights of SOS constraint.

Synopsis

```
void set(
    VarArray vars,
    double[] weights,
    int type)
```

Arguments

`vars`: variable array object.

`weights`: array of weights.

`type`: type of SOS constraint.

20.2.16 SosBuilderArray

COPT SOS constraint builder array object. To store and access a set of Java *SosBuilder* objects, Cardinal Optimizer provides Java SosBuilderArray class, which defines the following methods.

SosBuilderArray.SosBuilderArray()

Constructor of sosbuilderarray object.

Synopsis

```
SosBuilderArray()
```

SosBuilderArray.getBuilder()

Get idx-th SOS constraint builder object.

Synopsis

```
SosBuilder getBuilder(int idx)
```

Arguments

`idx`: index of the SOS constraint builder.

Return

SOS constraint builder object with index idx.

SosBuilderArray.pushBack()

Add a SOS constraint builder object to SOS constraint builder array.

Synopsis

```
void pushBack(SosBuilder builder)
```

Arguments

`builder`: a SOS constraint builder object.

SosBuilderArray.size()

Get the number of SOS constraint builder objects.

Synopsis

```
int size()
```

Return

number of SOS constraint builder objects.

20.2.17 GenConstr

COPT general constraint object. General constraints are always associated with a particular model. User creates a general constraint object by adding a general constraint to a model, rather than by using constructor of GenConstr class.

GenConstr.getIdx()

Get the index of the general constraint.

Synopsis

```
int getIdx()
```

Return

index of the general constraint.

GenConstr.getIIS()

Get IIS status of the general constraint.

Synopsis

```
int getIIS()
```

Return

IIS status.

GenConstr.remove()

Remove the general constraint from model.

Synopsis

```
void remove()
```

20.2.18 GenConstrArray

COPT general constraint array object. To store and access a set of Java *GenConstr* objects, Cardinal Optimizer provides Java GenConstrArray class, which defines the following methods.

GenConstrArray.GenConstrArray()

Constructor of genconstrarray.

Synopsis

```
GenConstrArray()
```

GenConstrArray.getGenConstr()

Get idx-th general constraint object.

Synopsis

```
GenConstr getGenConstr(int idx)
```

Arguments

`idx`: index of the general constraint.

Return

general constraint object with index idx.

GenConstrArray.pushBack()

Add a general constraint object to general constraint array.

Synopsis

```
void pushBack(GenConstr genconstr)
```

Arguments

`genconstr`: a general constraint object.

GenConstrArray.size()

Get the number of general constraint objects.

Synopsis

```
int size()
```

Return

number of general constraint objects.

20.2.19 GenConstrBuilder

COPT general constraint builder object. To help building a general constraint, given a binary variable and associated value, a linear expression and constraint sense, Cardinal Optimizer provides Java GenConstrBuilder class, which defines the following methods.

GenConstrBuilder.GenConstrBuilder()

Constructor of genconstrbuilder.

Synopsis

```
GenConstrBuilder()
```

GenConstrBuilder.getBinVal()

Get binary value associated with general constraint.

Synopsis

```
int getBinVal()
```

Return

binary value.

GenConstrBuilder.getBinVar()

Get binary variable associated with general constraint.

Synopsis

```
Var getBinVar()
```

Return

binary vaiable object.

GenConstrBuilder.getExpr()

Get expression associated with general constraint.

Synopsis

```
Expr getExpr()
```

Return

expression object.

GenConstrBuilder.getSense()

Get sense associated with general constraint.

Synopsis

```
char getSense()
```

Return

constraint sense.

GenConstrBuilder.set()

Set binary variable, binary value, expression and sense of general constraint.

Synopsis

```
void set(
    Var binvar,
    int binval,
    Expr expr,
    char sense)
```

Arguments

`binvar`: binary variable.

`binval`: binary value.

`expr`: expression object.

`sense`: general constraint sense.

20.2.20 GenConstrBuilderArray

COPT general constraint builder array object. To store and access a set of Java *GenConstrBuilder* objects, Cardinal Optimizer provides Java GenConstrBuilderArray class, which defines the following methods.

GenConstrBuilderArray.GenConstrBuilderArray()

Constructor of genconstrbuilderarray.

Synopsis

```
GenConstrBuilderArray()
```

GenConstrBuilderArray.getBuilder()

Get idx-th general constraint builder object.

Synopsis

```
GenConstrBuilder getBuilder(int idx)
```

Arguments

`idx`: index of the general constraint builder.

Return

general constraint builder object with index idx.

GenConstrBuilderArray.pushBack()

Add a general constraint builder object to general constraint builder array.

Synopsis

```
void pushBack(GenConstrBuilder builder)
```

Arguments

`builder`: a general constraint builder object.

GenConstrBuilderArray.size()

Get the number of general constraint builder objects.

Synopsis

```
int size()
```

Return

number of general constraint builder objects.

20.2.21 Cone

COPT cone constraint object. Cone constraints are always associated with a particular model. User creates a cone constraint object by adding a cone constraint to a model, rather than by using constructor of Cone class.

A cone constraint can be regular or rotated (`COPT_CONE_QUAD` or `COPT_CONE_RQUAD`).

Cone.getIdx()

Get the index of a cone constarint.

Synopsis

```
int getIdx()
```

Return

index of a cone constraint.

Cone.remove()

Remove the cone constraint from model.

Synopsis

```
void remove()
```

20.2.22 ConeArray

COPT cone constraint array object. To store and access a set of Java *Cone* objects, Cardinal Optimizer provides Java ConeArray class, which defines the following methods.

ConeArray.ConeArray()

Constructor of conearray object.

Synopsis

```
ConeArray()
```

ConeArray.getCone()

Get idx-th cone object.

Synopsis

```
Cone getCone(int idx)
```

Arguments

`idx`: index of cone.

Return

cone object with index idx.

ConeArray.pushBack()

Add a cone constraint object to cone constraint array.

Synopsis

```
void pushBack(Cone cone)
```

Arguments

`cone`: a cone constraint object.

ConeArray.size()

Get the number of cone constraint objects.

Synopsis

```
int size()
```

Return

number of cone constraint objects.

20.2.23 ConeBuilder

COPT cone constraint builder object. To help building a cone constraint, given the cone type and a set of variables, Cardinal Optimizer provides Java ConeBuilder class, which defines the following methods.

ConeBuilder.ConeBuilder()

Constructor of conebuilder object.

Synopsis

```
ConeBuilder()
```

ConeBuilder.getSize()

Get number of variables in a cone constraint.

Synopsis

```
int getSize()
```

Return

number of vars.

ConeBuilder.getType()

Get type of a cone constraint.

Synopsis

```
int getType()
```

Return

type of a cone constraint.

ConeBuilder.getVar()

Get idx-th variable in a cone constraint.

Synopsis

```
Var getVar(int idx)
```

Arguments

`idx`: index of variables.

Return

the idx-th variable in a cone constraint.

ConeBuilder.getVars()

Get all variables in a cone constraint.

Synopsis

```
VarArray getVars()
```

Return

variables in a cone constraint.

ConeBuilder.set()

Set variables and type of a cone constraint.

Synopsis

```
void set(VarArray vars, int type)
```

Arguments

`vars`: variable array object.

`type`: type of a cone constraint.

20.2.24 ConeBuilderArray

COPT cone constraint builder array object. To store and access a set of Java *ConeBuilder* objects, Cardinal Optimizer provides Java ConeBuilderArray class, which defines the following methods.

ConeBuilderArray.ConeBuilderArray()

Constructor of conebuilderarray object.

Synopsis

```
ConeBuilderArray()
```

ConeBuilderArray.getBuilder()

Get idx-th cone constraint builder object.

Synopsis

```
ConeBuilder getBuilder(int idx)
```

Arguments

`idx`: index of the cone constraint builder.

Return

cone constraint builder object with index idx.

ConeBuilderArray.pushBack()

Add a cone constraint builder object to cone constraint builder array.

Synopsis

```
void pushBack(ConeBuilder builder)
```

Arguments

`builder`: a cone constraint builder object.

ConeBuilderArray.size()

Get the number of cone constraint builder objects.

Synopsis

```
int size()
```

Return

number of cone constraint builder objects.

20.2.25 QuadExpr

COPT quadratic expression object. A quadratic expression consists of a linear expression, a list of variable pairs and associated coefficients of quadratic terms. Quadratic expressions are used to build quadratic constraints and objectives.

QuadExpr.QuadExpr()

Constructor of a constant quadratic expression with constant 0.0

Synopsis

```
QuadExpr()
```

QuadExpr.QuadExpr()

Constructor of a constant quadratic expression.

Synopsis

```
QuadExpr(double constant)
```

Arguments

`constant`: constant value in expression object.

QuadExpr.QuadExpr()

Constructor of a quadratic expression with one term.

Synopsis

```
QuadExpr(Var var)
```

Arguments

`var`: variable for the added term.

QuadExpr.QuadExpr()

Constructor of a quadratic expression with one term.

Synopsis

```
QuadExpr(Var var, double coeff)
```

Arguments

`var`: variable for the added term.

`coeff`: coefficent for the added term.

QuadExpr.QuadExpr()

Constructor of a quadratic expression with a linear expression.

Synopsis

```
QuadExpr(Expr expr)
```

Arguments

`expr`: linear expression added to the quadratic expression.

QuadExpr.QuadExpr()

Constructor of a quadratic expression with two linear expression.

Synopsis

```
QuadExpr(Expr expr, Var var)
```

Arguments

`expr`: one linear expression.

`var`: another variable.

QuadExpr.QuadExpr()

Constructor of a quadratic expression with two linear expression.

Synopsis

```
QuadExpr(Expr left, Expr right)
```

Arguments

`left`: one linear expression.

`right`: another linear expression.

QuadExpr.addConstant()

Add a constant to the quadratic expression.

Synopsis

```
void addConstant(double constant)
```

Arguments

`constant`: value to be added.

QuadExpr.addLinExpr()

Add a linear expression to self.

Synopsis

```
void addLinExpr(Expr expr)
```

Arguments

`expr`: linear expression to be added.

QuadExpr.addLinExpr()

Add a linear expression to self.

Synopsis

```
void addLinExpr(Expr expr, double mult)
```

Arguments

`expr`: linear expression to be added.

`mult`: multiplier constant.

QuadExpr.addQuadExpr()

Add a quadratic expression to self.

Synopsis

```
void addQuadExpr(QuadExpr expr)
```

Arguments

`expr`: quadratic expression to be added.

QuadExpr.addQuadExpr()

Add a quadratic expression to self.

Synopsis

```
void addQuadExpr(QuadExpr expr, double mult)
```

Arguments

`expr`: quadratic expression to be added.

`mult`: multiplier constant.

QuadExpr.addTerm()

Add a term to quadratic expression object.

Synopsis

```
void addTerm(Var var, double coeff)
```

Arguments

`var`: a variable of new term.

`coeff`: coefficient of new term.

QuadExpr.addTerm()

Add a quadratic term to expression object.

Synopsis

```
void addTerm(
    Var var1,
    Var var2,
    double coeff)
```

Arguments

`var1`: first variable of new quadratic term.

`var2`: second variable of new quadratic term.

`coeff`: coefficient of new quadratic term.

QuadExpr.addTerms()

Add linear terms to quadratic expression object.

Synopsis

```
void addTerms(Var[] vars, double coeff)
```

Arguments

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

QuadExpr.addTerms()

Add linear terms to quadratic expression object.

Synopsis

```
void addTerms(Var[] vars, double[] coeffs)
```

Arguments

`vars`: variables of added linear terms.

`coeffs`: coefficients of added linear terms.

QuadExpr.addTerms()

Add linear terms to quadratic expression object.

Synopsis

```
void addTerms(VarArray vars, double coeff)
```

Arguments

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

QuadExpr.addTerms()

Add linear terms to quadratic expression object.

Synopsis

```
void addTerms(VarArray vars, double[] coeffs)
```

Arguments

`vars`: variables of added terms.

`coeffs`: coefficients of added terms.

QuadExpr.addTerms()

Add quadratic terms to expression object.

Synopsis

```
void addTerms(
    VarArray vars1,
    VarArray vars2,
    double[] coeffs)
```

Arguments

`vars1`: first set of variables for added quadratic terms.

`vars2`: second set of variables for added quadratic terms.

`coeffs`: coefficient array for added quadratic terms.

QuadExpr.addTerms()

Add quadratic terms to expression object.

Synopsis

```
void addTerms(
    Var[] vars1,
    Var[] vars2,
    double[] coeffs)
```

Arguments

`vars1`: first set of variables for added quadratic terms.

`vars2`: second set of variables for added quadratic terms.

`coeffs`: coefficient array for added quadratic terms.

QuadExpr.clone()

Deep copy quadratic expression object.

Synopsis

```
QuadExpr clone()
```

Return

cloned quadratic expression object.

QuadExpr.evaluate()

evaluate quadratic expression after solving

Synopsis

```
double evaluate()
```

Return

value of quadratic expression

QuadExpr.getCoeff()

Get coefficient from the i-th term in quadratic expression.

Synopsis

```
double getCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term in quadratic expression object.

QuadExpr.getConstant()

Get constant in quadratic expression.

Synopsis

```
double getConstant()
```

Return

constant in quadratic expression.

QuadExpr.getLinExpr()

Get linear expression in quadratic expression.

Synopsis

```
Expr getLinExpr()
```

Return

linear expression object.

QuadExpr.getVar1()

Get first variable from the i-th term in quadratic expression.

Synopsis

```
Var getVar1(int i)
```

Arguments

`i`: index of the term.

Return

first variable of the i-th term in quadratic expression object.

QuadExpr.getVar2()

Get second variable from the i-th term in quadratic expression.

Synopsis

```
Var getVar2(int i)
```

Arguments

`i`: index of the term.

Return

second variable of the i-th term in quadratic expression object.

QuadExpr.remove()

Remove idx-th term from quadratic expression object.

Synopsis

```
void remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

QuadExpr.remove()

Remove the term associated with variable from quadratic expression.

Synopsis

```
void remove(Var var)
```

Arguments

`var`: a variable whose term should be removed.

QuadExpr.setCoeff()

Set coefficient of the i-th term in quadratic expression.

Synopsis

```
void setCoeff(int i, double val)
```

Arguments

`i`: index of the quadratic term.

`val`: coefficient of the term.

QuadExpr.setConstant()

Set constant for the quadratic expression.

Synopsis

```
void setConstant(double constant)
```

Arguments

`constant`: the value of the constant.

QuadExpr.size()

Get number of terms in quadratic expression.

Synopsis

```
long size()
```

Return

number of quadratic terms.

20.2.26 QConstraint

COPT quadratic constraint object. Quadratic constraints are always associated with a particular model. User creates a quadratic constraint object by adding a quadratic constraint to a model, rather than by using constructor of QConstraint class.

QConstraint.get()

Get attribute value of the quadratic constraint.

Synopsis

```
double get(String attr)
```

Arguments

`attr`: name of the attribute being queried.

Return

attribute value.

QConstraint.getIdx()

Get index of the quadratic constraint.

Synopsis

```
int getIdx()
```

Return

the index of the quadratic constraint.

QConstraint.getName()

Get name of the constraint.

Synopsis

```
String getName()
```

Return

the name of the constraint.

QConstraint.getRhs()

Get rhs of quadratic constraint.

Synopsis

```
double getRhs()
```

Return

rhs of quadratic constraint.

QConstraint.getSense()

Get rhs of quadratic constraint.

Synopsis

```
char getSense()
```

Return

rhs of quadratic constraint.

QConstraint.remove()

Remove this constraint from model.

Synopsis

```
void remove()
```

QConstraint.set()

Set attribute value of the quadratic constraint.

Synopsis

```
void set(String attr, double val)
```

Arguments

`attr`: name of the attribute.

`val`: new value.

QConstraint.setName()

Set name of quadratic constraint.

Synopsis

```
void setName(String name)
```

Arguments

`name`: the name to set.

QConstraint.setRhs()

Set rhs of quadratic constraint.

Synopsis

```
void setRhs(double rhs)
```

Arguments

`rhs`: rhs of quadratic constraint.

QConstraint.setSense()

Set sense of quadratic constraint.

Synopsis

```
void setSense(char sense)
```

Arguments

`sense`: sense of quadratic constraint.

20.2.27 QConstrArray

COPT quadratic constraint array object. To store and access a set of Java *QConstraint* objects, Cardinal Optimizer provides Java QConstrArray class, which defines the following methods.

QConstrArray.QConstrArray()

QConstructor of constrarray object.

Synopsis

```
QConstrArray()
```

QConstrArray.getQConstr()

Get idx-th constraint object.

Synopsis

```
QConstraint getQConstr(int idx)
```

Arguments

`idx`: index of the constraint.

Return

constraint object with index idx.

QConstrArray.pushBack()

Add a constraint object to constraint array.

Synopsis

```
void pushBack(QConstraint constr)
```

Arguments

`constr`: a constraint object.

QConstrArray.size()

Get the number of constraint objects.

Synopsis

```
int size()
```

Return

number of constraint objects.

20.2.28 QConstrBuilder

COPT quadratic constraint builder object. To help building a quadratic constraint, given a quadratic expression, constraint sense and right-hand side value, Cardinal Optimizer provides Java ConeBuilder class, which defines the following methods.

QConstrBuilder.QConstrBuilder()

QConstructor of constrbuilder object.

Synopsis

```
QConstrBuilder()
```

QConstrBuilder.getQuadExpr()

Get expression associated with constraint.

Synopsis

```
QuadExpr getQuadExpr()
```

Return

quadratic expression object.

QConstrBuilder.getSense()

Get sense associated with quadratic constraint.

Synopsis

```
char getSense()
```

Return

quadratic constraint sense.

QConstrBuilder.set()

Set detail of a quadratic constraint to its builder object.

Synopsis

```
void set(
    QuadExpr expr,
    char sense,
    double rhs)
```

Arguments

`expr`: expression object at one side of the quadratic constraint.

`sense`: quadratic constraint sense.

`rhs`: constant of right side of quadratic constraint.

20.2.29 QConstrBuilderArray

COPT quadratic constraint builder array object. To store and access a set of Java *QConstrBuilder* objects, Cardinal Optimizer provides Java QConstrBuilderArray class, which defines the following methods.

QConstrBuilderArray.QConstrBuilderArray()

QConstructor of constrbuilderarray object.

Synopsis

```
QConstrBuilderArray()
```

QConstrBuilderArray.getBuilder()

Get idx-th constraint builder object.

Synopsis

```
QConstrBuilder getBuilder(int idx)
```

Arguments

`idx`: index of the constraint builder.

Return

constraint builder object with index idx.

QConstrBuilderArray.pushBack()

Add a constraint builder object to constraint builder array.

Synopsis

```
void pushBack(QConstrBuilder builder)
```

Arguments

`builder`: a constraint builder object.

QConstrBuilderArray.size()

Get the number of constraint builder objects.

Synopsis

```
int size()
```

Return

number of constraint builder objects.

20.2.30 PsdVar

COPT PSD variable object. PSD variables are always associated with a particular model. User creates a PSD variable object by adding a PSD variable to model, rather than by constructor of PsdVar class.

PsdVar.get()

Get attribute values of PSD variable.

Synopsis

```
double[] get(String attr)
```

Arguments

`attr`: attribute name.

Return

output array of attribute values.

PsdVar.getDim()

Get dimension of PSD variable.

Synopsis

```
int getDim()
```

Return

dimension of PSD variable.

PsdVar.getIdx()

Get length of PSD variable.

Synopsis

```
int getIdx()
```

Return

length of PSD variable.

PsdVar.getLen()

Get length of PSD variable.

Synopsis

```
int getLen()
```

Return

length of PSD variable.

PsdVar.getName()

Get name of PSD variable.

Synopsis

```
String getName()
```

Return

name of PSD variable.

PsdVar.remove()

Remove PSD variable from model.

Synopsis

```
void remove()
```

20.2.31 PsdVarArray

COPT PSD variable array object. To store and access a set of *PsdVar* objects, Cardinal Optimizer provides PsdVarArray class, which defines the following methods.

PsdVarArray.PsdVarArray()

Constructor of PsdVarArray.

Synopsis

```
PsdVarArray()
```

PsdVarArray.getPsdVar()

Get idx-th PSD variable object.

Synopsis

```
PsdVar getPsdVar(int idx)
```

Arguments

`idx`: index of the PSD variable.

Return

PSD variable object with index idx.

PsdVarArray.pushBack()

Add a PSD variable object to PSD variable array.

Synopsis

```
void pushBack(PsdVar var)
```

Arguments

`var`: a PSD variable object.

PsdVarArray.reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void reserve(int n)
```

Arguments

`n`: minimum capacity for PSD variable object.

PsdVarArray.size()

Get the number of PSD variable objects.

Synopsis

```
int size()
```

Return

number of PSD variable objects.

20.2.32 PsdExpr

COPT PSD expression object. A PSD expression consists of a linear expression, a list of PSD variables and associated coefficient matrices of PSD terms. PSD expressions are used to build PSD constraints and objectives.

PsdExpr.PsdExpr()

Constructor of a PSD expression with default constant value 0.

Synopsis

```
PsdExpr(double constant)
```

Arguments

`constant`: optional, constant value in PSD expression object.

PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

Synopsis

```
PsdExpr(Var var)
```

Arguments

`var`: variable for the added term.

PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

Synopsis

```
PsdExpr(Var var, double coeff)
```

Arguments

`var`: variable for the added term.

`coeff`: coefficent for the added term.

PsdExpr.PsdExpr()

Constructor of a PSD expression with a linear expression.

Synopsis

```
PsdExpr(Expr expr)
```

Arguments

`expr`: input linear expression.

PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

Synopsis

```
PsdExpr(PsdVar var, SymMatrix mat)
```

Arguments

`var`: PSD variable for the added term.

`mat`: coefficient matrix for the added term.

PsdExpr.PsdExpr()

Constructor of a PSD expression with one term.

Synopsis

```
PsdExpr(PsdVar var, SymMatExpr expr)
```

Arguments

`var`: PSD variable for the added term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

PsdExpr.addConstant()

Add constant to the PSD expression.

Synopsis

```
void addConstant(double constant)
```

Arguments

`constant`: value to be added.

PsdExpr.addLinExpr()

Add a linear expression to PSD expression object.

Synopsis

```
void addLinExpr(Expr expr)
```

Arguments

`expr`: linear expression to be added.

PsdExpr.addLinExpr()

Add a linear expression to PSD expression object.

Synopsis

```
void addLinExpr(Expr expr, double mult)
```

Arguments

`expr`: linear expression to be added.

`mult`: multiplier constant.

PsdExpr.addPsdExpr()

Add a PSD expression to self.

Synopsis

```
void addPsdExpr(PsdExpr expr)
```

Arguments

`expr`: PSD expression to be added.

PsdExpr.addPsdExpr()

Add a PSD expression to self.

Synopsis

```
void addPsdExpr(PsdExpr expr, double mult)
```

Arguments

`expr`: PSD expression to be added.

`mult`: multiplier constant.

PsdExpr.addTerm()

Add a linear term to PSD expression object.

Synopsis

```
void addTerm(Var var, double coeff)
```

Arguments

`var`: variable of new linear term.

`coeff`: coefficient of new linear term.

PsdExpr.addTerm()

Add a PSD term to PSD expression object.

Synopsis

```
void addTerm(PsdVar var, SymMatrix mat)
```

Arguments

`var`: PSD variable of new PSD term.

`mat`: coefficient matrix of new PSD term.

PsdExpr.addTerm()

Add a PSD term to PSD expression object.

Synopsis

```
void addTerm(PsdVar var, SymMatExpr expr)
```

Arguments

`var`: PSD variable of new PSD term.

`expr`: coefficient expression of symmetric matrices of new PSD term.

PsdExpr.addTerms()

Add linear terms to PSD expression object.

Synopsis

```
void addTerms(Var[] vars, double coeff)
```

Arguments

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

PsdExpr.addTerms()

Add linear terms to PSD expression object.

Synopsis

```
void addTerms(Var[] vars, double[] coeffs)
```

Arguments

`vars`: variables for added linear terms.

`coeffs`: coefficient array for added linear terms.

PsdExpr.addTerms()

Add linear terms to PSD expression object.

Synopsis

```
void addTerms(VarArray vars, double coeff)
```

Arguments

`vars`: variables of added linear terms.

`coeff`: one coefficient for added linear terms.

PsdExpr.addTerms()

Add linear terms to PSD expression object.

Synopsis

```
void addTerms(VarArray vars, double[] coeffs)
```

Arguments

`vars`: variables of added terms.

`coeffs`: coefficients of added terms.

PsdExpr.addTerms()

Add PSD terms to PSD expression object.

Synopsis

```
void addTerms(PsdVarArray vars, SymMatrixArray mats)
```

Arguments

`vars`: PSD variables for added PSD terms.

`mats`: coefficient matrixes for added PSD terms.

PsdExpr.addTerms()

Add PSD terms to PSD expression object.

Synopsis

```
void addTerms(PsdVar[] vars, SymMatrix[] mats)
```

Arguments

`vars`: PSD variables for added PSD terms.

`mats`: coefficient matrixes for added PSD terms.

PsdExpr.clone()

Deep copy PSD expression object.

Synopsis

```
PsdExpr clone()
```

Return

cloned PSD expression object.

PsdExpr.evaluate()

Evaluate PSD expression after solving

Synopsis

```
double evaluate()
```

Return

Value of PSD expression

PsdExpr.getCoeff()

Get coefficient from the i-th term in PSD expression.

Synopsis

```
SymMatExpr getCoeff(int i)
```

Arguments

`i`: index of the PSD term.

Return

coefficient expression of the i-th PSD term.

PsdExpr.getConstant()

Get constant in PSD expression.

Synopsis

```
double getConstant()
```

Return

constant in PSD expression.

PsdExpr.getLinExpr()

Get linear expression in PSD expression.

Synopsis

```
Expr getLinExpr()
```

Return

linear expression object.

PsdExpr.getPsdVar()

Get the PSD variable from the i-th term in PSD expression.

Synopsis

```
PsdVar getPsdVar(int i)
```

Arguments

`i`: index of the term.

Return

the first variable of the i-th term in PSD expression object.

PsdExpr.multiply()

Multiply a PSD expression and a constant.

Synopsis

```
void multiply(double c)
```

Arguments

`c`: constant operand.

PsdExpr.remove()

Remove i-th term from PSD expression object.

Synopsis

```
void remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

PsdExpr.remove()

Remove the term associated with variable from PSD expression.

Synopsis

```
void remove(Var var)
```

Arguments

`var`: a variable whose term should be removed.

PsdExpr.remove()

Remove the term associated with PSD variable from PSD expression.

Synopsis

```
void remove(PsdVar var)
```

Arguments

`var`: a PSD variable whose term should be removed.

PsdExpr.setCoeff()

Set coefficient matrix of the i-th term in PSD expression.

Synopsis

```
void setCoeff(int i, SymMatrix mat)
```

Arguments

`i`: index of the PSD term.

`mat`: coefficient matrix of the term.

PsdExpr.setConstant()

Set constant for the PSD expression.

Synopsis

```
void setConstant(double constant)
```

Arguments

`constant`: the value of the constant.

PsdExpr.size()

Get number of PSD terms in expression.

Synopsis

```
long size()
```

Return

number of PSD terms.

20.2.33 PsdConstraint

COPT PSD constraint object. PSD constraints are always associated with a particular model. User creates a PSD constraint object by adding a PSD constraint to model, rather than by constructor of PsdConstraint class.

PsdConstraint.get()

Get attribute value of the PSD constraint. Support related PSD attributes.

Synopsis

```
double get(String attr)
```

Arguments

`attr`: name of queried attribute.

Return

attribute value.

PsdConstraint.getIdx()

Get index of the PSD constraint.

Synopsis

```
int getIdx()
```

Return

the index of the PSD constraint.

PsdConstraint.getName()

Get name of the PSD constraint.

Synopsis

```
String getName()
```

Return

the name of the PSD constraint.

PsdConstraint.remove()

Remove this PSD constraint from model.

Synopsis

```
void remove()
```

PsdConstraint.set()

Set attribute value of the PSD constraint. Support related PSD attributes.

Synopsis

```
void set(String attr, double value)
```

Arguments

`attr`: name of queried attribute.

`value`: new value.

PsdConstraint.setName()

Set name of a PSD constraint.

Synopsis

```
void setName(String name)
```

Arguments

`name`: the name to set.

20.2.34 PsdConstrArray

COPT PSD constraint array object. To store and access a set of *PsdConstraint* objects, Cardinal Optimizer provides PsdConstrArray class, which defines the following methods.

PsdConstrArray.PsdConstrArray()

Constructor of PsdConstrArray object.

Synopsis

```
PsdConstrArray()
```

PsdConstrArray.getPsdConstr()

Get idx-th PSD constraint object.

Synopsis

```
PsdConstraint getPsdConstr(int idx)
```

Arguments

`idx`: index of the PSD constraint.

Return

PSD constraint object with index idx.

PsdConstrArray.pushBack()

Add a PSD constraint object to PSD constraint array.

Synopsis

```
void pushBack(PsdConstraint constr)
```

Arguments

`constr`: a PSD constraint object.

PsdConstrArray.reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void reserve(int n)
```

Arguments

`n`: minimum capacity for PSD constraint objects.

PsdConstrArray.size()

Get the number of PSD constraint objects.

Synopsis

```
int size()
```

Return

number of PSD constraint objects.

20.2.35 PsdConstrBuilder

COPT PSD constraint builder object. To help building a PSD constraint, given a PSD expression, constraint sense and right-hand side value, Cardinal Optimizer provides PsdConstrBuilder class, which defines the following methods.

PsdConstrBuilder.PsdConstrBuilder()

Constructor of PsdConstrBuilder object.

Synopsis

```
PsdConstrBuilder()
```

PsdConstrBuilder.getPsdExpr()

Get expression associated with PSD constraint.

Synopsis

```
PsdExpr getPsdExpr()
```

Return

PSD expression object.

PsdConstrBuilder.getRange()

Get range from lower bound to upper bound of range constraint.

Synopsis

```
double getRange()
```

Return

length from lower bound to upper bound of the constraint.

PsdConstrBuilder.getSense()

Get sense associated with PSD constraint.

Synopsis

```
char getSense()
```

Return

PSD constraint sense.

PsdConstrBuilder.set()

Set detail of a PSD constraint to its builder object.

Synopsis

```
void set(
    PsdExpr expr,
    char sense,
    double rhs)
```

Arguments

`expr`: expression object at one side of the PSD constraint.

`sense`: PSD constraint sense, other than COPT_RANGE.

`rhs`: constant at right side of the PSD constraint.

PsdConstrBuilder.setRange()

Set a range constraint to its builder.

Synopsis

```
void setRange(PsdExpr expr, double range)
```

Arguments

`expr`: PSD expression object, whose constant is negative upper bound.

`range`: length from lower bound to upper bound of the constraint. Must greater than 0.

20.2.36 PsdConstrBuilderArray

COPT PSD constraint builder array object. To store and access a set of *PsdConstrBuilder* objects, Cardinal Optimizer provides PsdConstrBuilderArray class, which defines the following methods.

PsdConstrBuilderArray.PsdConstrBuilderArray()

Constructor of PsdConstrBuilderArray object.

Synopsis

```
PsdConstrBuilderArray()
```

PsdConstrBuilderArray.getBuilder()

Get idx-th PSD constraint builder object.

Synopsis

```
PsdConstrBuilder getBuilder(int idx)
```

Arguments

`idx`: index of the PSD constraint builder.

Return

PSD constraint builder object with index idx.

PsdConstrBuilderArray.pushBack()

Add a PSD constraint builder to PSD constraint builder array.

Synopsis

```
void pushBack(PsdConstrBuilder builder)
```

Arguments

`builder`: a PSD constraint builder object.

PsdConstrBuilderArray.reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void reserve(int n)
```

Arguments

`n`: minimum capacity for PSD constraint builder object.

PsdConstrBuilderArray.size()

Get the number of PSD constraint builder objects.

Synopsis

```
int size()
```

Return

number of PSD constraint builder objects.

20.2.37 SymMatrix

COPT symmetric matrix object. Symmetric matrices are always associated with a particular model. User creates a symmetric matrix object by adding a symmetric matrix to model, rather than by constructor of SymMatrix class.

Symmetric matrices are used as coefficient matrices of PSD terms in PSD expressions, PSD constraints or PSD objectives.

SymMatrix.getDim()

Get the dimension of a symmetric matrix.

Synopsis

```
int getDim()
```

Return

Dimension of a symmetric matrix.

SymMatrix.getIdx()

Get the index of a symmetric matrix.

Synopsis

```
int getIdx()
```

Return

Index of a symmetric matrix.

20.2.38 SymMatrixArray

COPT symmetric matrix object. To store and access a set of *SymMatrix* objects, Cardinal Optimizer provides SymMatrixArray class, which defines the following methods.

SymMatrixArray.SymMatrixArray()

Constructor of SymMatrixAarray.

Synopsis

```
SymMatrixArray()
```

SymMatrixArray.getMatrix()

Get i-th SymMatrix object.

Synopsis

```
SymMatrix getMatrix(int idx)
```

Arguments

`idx`: index of the SymMatrix object.

Return

SymMatrix object with index idx.

SymMatrixArray.pushBack()

Add a SymMatrix object to SymMatrix array.

Synopsis

```
void pushBack(SymMatrix mat)
```

Arguments

`mat`: a SymMatrix object.

SymMatrixArray.reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void reserve(int n)
```

Arguments

`n`: minimum capacity for symmetric matrix object.

SymMatrixArray.size()

Get the number of SymMatrix objects.

Synopsis

```
int size()
```

Return

number of SymMatrix objects.

20.2.39 SymMatExpr

COPT symmetric matrix expression object. A symmetric matrix expression is a linear combination of symmetric matrices, which is still a symmetric matrix. However, by doing so, we are able to delay computing the final matrix until setting PSD constraints or PSD objective.

SymMatExpr.SymMatExpr()

Constructor of a symmetric matrix expression.

Synopsis

```
SymMatExpr()
```

SymMatExpr.SymMatExpr()

Constructor of a symmetric matrix expression with one term.

Synopsis

```
SymMatExpr(SymMatrix mat, double coeff)
```

Arguments

`mat`: symmetric matrix of the added term.

`coeff`: coefficent for the added term.

SymMatExpr.addSymMatExpr()

Add a symmetric matrix expression to self.

Synopsis

```
void addSymMatExpr(SymMatExpr expr, double mult)
```

Arguments

`expr`: symmetric matrix expression to be added.

`mult`: constant multiplier.

SymMatExpr.addTerm()

Add a term to symmetric matrix expression object.

Synopsis

```
Boolean addTerm(SymMatrix mat, double coeff)
```

Arguments

`mat`: symmetric matrix of the new term.

`coeff`: coefficient of the new term.

Return

True if the term is added successfully.

SymMatExpr.addTerms()

Add multiple terms to expression object.

Synopsis

```
int addTerms(SymMatrixArray mats, double[] coeffs)
```

Arguments

`mats`: symmetric matrix array object for added terms.

`coeffs`: coefficient array for added terms.

Return

Number of added terms. If negative, fail to add one of terms.

SymMatExpr.addTerms()

Add multiple terms to expression object.

Synopsis

```
int addTerms(SymMatrix[] mats, double[] coeffs)
```

Arguments

`mats`: symmetric matrix array object for added terms.

`coeffs`: coefficient array for added terms.

Return

Number of added terms. If negative, fail to add one of terms.

SymMatExpr.addTerms()

Add multiple terms to expression object.

Synopsis

```
int addTerms(SymMatrix[] mats, double coeff)
```

Arguments

`mats`: symmetric matrix array object for added terms.

`coeff`: common coefficient for added terms.

Return

Number of added terms. If negative, fail to add one of terms.

SymMatExpr.clone()

Deep copy symmetric matrix expression object.

Synopsis

```
SymMatExpr clone()
```

Return

cloned expression object.

SymMatExpr.getCoeff()

Get coefficient of the i-th term in expression object.

Synopsis

```
double getCoeff(int i)
```

Arguments

`i`: index of the term.

Return

coefficient of the i-th term.

SymMatExpr.getDim()

Get dimension of symmetric matrix in expression.

Synopsis

```
int getDim()
```

Return

dimension of symmetric matrix.

SymMatExpr.getSymMat()

Get symmetric matrix of the i-th term in expression object.

Synopsis

```
SymMatrix getSymMat(int i)
```

Arguments

`i`: index of the term.

Return

the symmetric matrix of the i-th term.

SymMatExpr.multiply()

multiply a symmetric matrix expression and a constant.

Synopsis

```
void multiply(double c)
```

Arguments

`c`: constant operand.

SymMatExpr.remove()

Remove i-th term from expression object.

Synopsis

```
void remove(int idx)
```

Arguments

`idx`: index of the term to be removed.

SymMatExpr.remove()

Remove the term associated with the symmetric matrix.

Synopsis

```
void remove(SymMatrix mat)
```

Arguments

`mat`: a symmetric matrix whose term should be removed.

SymMatExpr.reserve()

Reserve capacity to contain at least n items.

Synopsis

```
void reserve(int n)
```

Arguments

`n`: minimum capacity for expression object.

SymMatExpr.setCoeff()

Set coefficient for the i-th term in expression object.

Synopsis

```
void setCoeff(int i, double val)
```

Arguments

`i`: index of the term.

`val`: coefficient of the term.

SymMatExpr.size()

Get number of terms in expression.

Synopsis

```
long size()
```

Return

number of terms.

20.2.40 CallbackBase

COPT Callback abstract base object. Users must implment its virtual method `virtual void CallbackBase::callback()` to instantiate an instance, which pass to `Model::SetCallback(CallbackBase cb, int cbctx)` as the first parameter. Subclass of CallbackBase inherits the following member methods:

CallbackBase.CallbackBase()

Constructor of CallbackBase, implementing ICallback interface.

Synopsis

```
CallbackBase()
```

CallbackBase.addLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void addLazyConstr(
    Expr lhs,
    char sense,
    double rhs)
```

Arguments

`lhs`: expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side value for lazy constraint.

CallbackBase.addLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void addLazyConstr(
    Expr lhs,
    char sense,
    Expr rhs)
```

Arguments

`lhs`: left hand side expression for lazy contraint.

`sense`: sense for lazy constraint.

`rhs`: right hand side expression for lazy contraint.

CallbackBase.addLazyConstr()

Add a lazy constraint to model.

Synopsis

```
void addLazyConstr(ConstrBuilder builder)
```

Arguments

`builder`: builder for lazy contraint.

CallbackBase.addLazyConstrs()

Add lazy constraints to model.

Synopsis

```
void addLazyConstrs(ConstrBuilderArray builders)
```

Arguments

`builders`: array of builders for lazy contraints.

CallbackBase.addUserCut()

Add a user cut to model.

Synopsis

```
void addUserCut(
    Expr lhs,
    char sense,
    double rhs)
```

Arguments

`lhs`: expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side value for user cut.

CallbackBase.addUserCut()

Add a user cut to model.

Synopsis

```
void addUserCut(
    Expr lhs,
    char sense,
    Expr rhs)
```

Arguments

`lhs`: left hand side expression for user cut.

`sense`: sense for user cut.

`rhs`: right hand side expression for user cut.

CallbackBase.addUserCut()

Add a user cut to model.

Synopsis

```
void addUserCut(ConstrBuilder builder)
```

Arguments

`builder`: builder for user cut.

CallbackBase.addUserCuts()

Add user cuts to model.

Synopsis

```
void addUserCuts(ConstrBuilderArray builders)
```

Arguments

`builders`: array of builders for user cuts.

CallbackBase.callback()

Pure virtual function defined in ICallback interface. User must implement it.

Synopsis

```
void callback()
```

CallbackBase.getDblInfo()

Get double value of given information name in callback.

Synopsis

```
double getDblInfo(String cbinfo)
```

Arguments

`cbinfo`: name of callback info.

Return

value of desired information.

CallbackBase.getIncumbent()

Get best feasible solution of given variable in callback.

Synopsis

```
double getIncumbent(Var var)
```

Arguments

`var`: given variable.

Return

best feasible solution of given variable.

CallbackBase.getIncumbent()

Get best feasible solution of variables in callback.

Synopsis

```
double[] getIncumbent(VarArray vars)
```

Arguments

`vars`: an array of variables.

Return

best feasible solution of desired variables.

CallbackBase.getIncumbent()

Get best feasible solution of variables in callback.

Synopsis

```
double[] getIncumbent(Var[] vars)
```

Arguments

`vars`: an array of variables.

Return

best feasible solution of desired variables.

CallbackBase.getIncumbent()

Get best feasible solution of all variables in callback.

Synopsis

```
double[] getIncumbent()
```

Return

best feasible solution of all variables.

CallbackBase.getIntInfo()

Get integer value of given information name in callback.

Synopsis

```
int getIntInfo(String cbinfo)
```

Arguments

`cbinfo`: name of callback info.

Return

value of desired information.

CallbackBase.getRelaxSol()

Get LP-relaxation solution of given variable in callback.

Synopsis

```
double getRelaxSol(Var var)
```

Arguments

`var`: given variable.

Return

LP-relaxation solution of given variable.

CallbackBase.getRelaxSol()

Get LP-relaxation solution of variables in callback.

Synopsis

```
double[] getRelaxSol(VarArray vars)
```

Arguments

`vars`: an array of variables.

Return

LP-relaxation solution of variables.

CallbackBase.getRelaxSol()

Get LP-relaxation solution of variables in callback.

Synopsis

```
double[] getRelaxSol(Var[] vars)
```

Arguments

`vars`: an array of variables.

Return

LP-relaxation solution of variables.

CallbackBase.getRelaxSol()

Get LP-relaxation solution of all variables in callback.

Synopsis

```
double[] getRelaxSol()
```

Return

LP-relaxation solution of all variables.

CallbackBase.getSolution()

Get solution of given variable in callback.

Synopsis

```
double getSolution(Var var)
```

Arguments

`var`: given variable.

Return

solution of given variable.

CallbackBase.getSolution()

Get solution of variables in callback.

Synopsis

```
double[] getSolution(VarArray vars)
```

Arguments

`vars`: an array of variables.

Return

solution of variables.

CallbackBase.getSolution()

Get solution of variables in callback.

Synopsis

```
double[] getSolution(Var[] vars)
```

Arguments

`vars`: an array of variables.

Return

solution of variables.

CallbackBase.getSolution()

Get solution of all variables in callback.

Synopsis

```
double[] getSolution()
```

Return

solution of all variables.

CallbackBase.interrupt()

Interrupt solving problems in callback

Synopsis

```
void interrupt()
```

CallbackBase.loadSolution()

Load customized solution to model.

Synopsis

```
double loadSolution()
```

Return

objective value of given solution.

CallbackBase.setSolution()

Set solution of a given variable in callback.

Synopsis

```
void setSolution(Var var, double val)
```

Arguments

`var`: a variable object.

`val`: double value.

CallbackBase.setSolution()

Set solution of variables in callback.

Synopsis

```
void setSolution(VarArray vars, double[] vals)
```

Arguments

`vars`: an array of variable objects.

`vals`: an array of double values.

CallbackBase.setSolution()

Set solution of variables in callback.

Synopsis

```
void setSolution(Var[] vars, double[] vals)
```

Arguments

`vars`: an array of variable objects.

`vals`: an array of double values.

CallbackBase.where()

Get context in callback.

Synopsis

```
int where()
```

Return

integer value of context.

20.2.41 ProbBuffer

Buffer object for COPT problem. ProbBuffer object holds the (MPS) problem in string format.

ProbBuffer.ProbBuffer()

Constructor of ProbBuffer object.

Synopsis

```
ProbBuffer(int sz)
```

Arguments

`sz`: initial size of the problem buffer.

ProbBuffer.getData()

Get string of problem in problem buffer.

Synopsis

```
String getData()
```

Return

string of problem in problem buffer.

ProbBuffer.resize()

Resize buffer to given size, and zero-ended.

Synopsis

```
void resize(int sz)
```

Arguments

`sz`: new buffer size.

ProbBuffer.size()

Get the size of problem buffer.

Synopsis

```
int size()
```

Return

size of problem buffer.

20.2.42 CoptException

Copt exception object.

CoptException.CoptException()

Constructor of coptexception.

Synopsis

```
CoptException(int code, String msg)
```

Arguments

`code`: error code for exception.

`msg`: error message for exception.

CoptException.getCode()

Get the error code associated with the exception.

Synopsis

```
int getCode()
```

Return

the error code.

Chapter 21

Python API Reference

The **Cardinal Optimizer** provides a Python API library. This chapter documents all COPT Python constants and API functions for python applications.

21.1 Constants

Python Constants are necessary to solve a problem using the Python interface. There are four types of constants defined in COPT Python API library. They are general constants, information constants, parameters and attributes.

21.1.1 General Constants

For the contents of Python general constants, see *General Constants*.

General constants are those commonly used in modeling, such as optimization directions, variable types, and solving status, etc. Users may refer to general constants with a `'COPT'` prefix. For instance, `COPT.VERSION_MAJOR` is the major version number of the **Cardinal Optimizer**.

21.1.2 Attributes

For the contents of Python attribute constants, see *Attributes*.

In the Python API, users may refer to attributes using a `'COPT.Attr'` prefix. For instance, `COPT.Attr.Cols` is the number of variables or columns in the model.

In the Python API, user can get the attribute value by specifying the attribute name. Attributes are mostly used in `Model.getAttr()` method to query properties of the model, please refer to *Python API: Model Class* for details. Here is an example:

- `Model.getAttr()`: `Model.getAttr("Cols")` obtain the number of variables or columns in the model.

21.1.3 Information

For the contents of Python API information class constants, see *Information*.

In the Python API, user can access the information through the `COPT.Info` prefix. For instance, `COPT.Info.Obj` is the objective function coefficients for variables (columns).

In the Python API, user can get or set the information value of the object by specifying the information name:

- Get the value of variable or constraint information: `Model.getInfo()` / `Var.getInfo()` / `Constraint.getInfo()`
- Set the value of variable or constraint information: `Model.setInfo()` / `Var.setInfo()` / `Constraint.setInfo()`

21.1.4 Callback Information

For the content of Python API callback information class constants, see *Callback Information*.

In the Python API, callback-related information constants are defined in the `CbInfo` class. User can access the callback information via `COPT.CbInfo.` prefix.

For instance, `COPT.CbInfo.BestObj` is the current best objective.

In the Python API, user can get the value of callback information by specifying the information name.

For instance, `CallbackBase.getInfo(COPT.CbInfo.BestObj)` : get the value of the current best objective.

21.1.5 Parameters

For the contents of Python API Parameters class constants, see *Parameters*.

Parameters control the operation of the **Cardinal Optimizer**. They can be modified before the optimization begins.

In the Python API, user can access parameters through the `COPT.Param` prefix. For instance, `COPT.Param.TimeLimit` is time limit in seconds of the optimization.

In the Python API, user can get and set the parameter value by specifying the parameter name. The provided functions are as follows, please refer to *Python API: Model Class* for details.

- Get detailed information of the specified parameter (current value/max/min): `Model.getParamInfo()`
- Get the current value of the specified parameter: `Model.getParam()`
- Set the specified parameter value: `Model.setParam()`

21.2 Python Modeling Classes

Python modeling classes are essential for the Python interface of Cardinal Optimizer. It provides plentiful easy-to-use methods to quickly build optimization models in complex practical scenarios. This section will explains these functions and their usage.

21.2.1 EnvrConfig Class

EnvrConfig object contains operations related to client configuration, and provides the following methods:

EnvrConfig()

Synopsis

```
EnvrConfig()
```

Description

Constructor of EnvrConfig class. This method creates and returns an *EnvrConfig Class* object.

Example

```
# Create client configuration
envconfig = EnvrConfig()
```

EnvrConfig.set()

Synopsis

```
set(name, value)
```

Description

Set client configuration.

Arguments

`name`

Name of configuration parameter.

`value`

Value of configuration parameter.

Example

```
# Set client configuration
envconfig.set(COPT.CLIENT_WAITTIME, 600)
```

21.2.2 Envr Class

Envr object contains operations related to COPT optimization environment, and provides the following methods:

Envr()

Synopsis

```
Envr(arg=None)
```

Description

Function for constructing Envr object. This method creates and returns an *Envr Class* object.

Arguments

`arg`

Path of license file or client configuration. Optional argument, defaults to `None`.

Example

```
# Create solving environment
env = Envr()
```

Envr.createModel()

Synopsis

`createModel(name="")`

Description

Create optimization model and return a *Model Class* object.

Arguments

`name`

The name of the Model object to be created. Optional, `""` by default.

Example

```
# Create optimization model
model = env.createModel("coptprob")
```

Envr.close()

Synopsis

`close()`

Description

Close connection to remote server.

Example

```
# Close connection to remote server
env.close()
```

21.2.3 Model Class

For easy access to model's attributes and optimization parameters, Model object provides methods such as `Model.Rows`. The full list of attributes can be found in *Attributes* section. For convenience, attributes can be accessed by their names in capital or lower case.

Note that for LP or MIP, both the objective value and the solution status can be accessed through `Model.objval` and `Model.status`.

For optimization parameters, they can be set in the form `"Model.Param.TimeLimit = 10"`. For details of the parameter names supported, please refer to *Parameters* section.

Class Model contains COPT model-related operations and provides the following methods:

Model.addVar()

Synopsis

```
addVar(lb=0.0, ub=COPT.INFINITY, obj=0.0, vtype=COPT.CONTINUOUS,
name="", column=None)
```

Description

Add a decision variable to model and return the added *Var Class* object.

Arguments

`lb`

Lower bound for new variable. Optional, 0.0 by default.

`ub`

Upper bound for new variable. Optional, `COPT.INFINITY` by default.

`obj`

Objective parameter for new variable. Optional, 0.0 by default.

`vtype`

Variable type. Optional, `COPT.CONTINUOUS` by default. Please refer to Variable types for possible types.

`name`

Name for new variable. Optional, `""` by default, which is automatically generated by solver.

`column`

Column corresponds to the variable. Optional, `None` by default.

Example

```
# Add a continuous variable
x = m.addVar()
# Add a binary variable
y = m.addVar(vtype=COPT.BINARY)
# Add an integer variable with lowerbound -1.0, upperbound 1.0, objective coefficient 1.0 and␣
↪variable name "z"
z = m.addVar(-1.0, 1.0, 1.0, COPT.INTEGER, "z")
```

Model.addVars()

Synopsis

```
addVars(*indices, lb=0.0, ub=COPT.INFINITY, obj=0.0, vtype=COPT.
CONTINUOUS, nameprefix="C")
```

Description

Add multiple new variables to a model. Return a *tupledict Class*, whose key is indice of the variable and value is the *Var Class* object.

Arguments

`*indices`

Indices for accessing the new variables.

`lb`

Lower bounds for new variables. Optional, 0.0 by default.

`ub`

Upper bounds for new variables. Optional, `COPT.INFINITY` by default.

`obj`

Objective costs for new variables. Optional, 0.0 by default.

`vtype`

Variable types. Optional, `COPT.CONTINUOUS` by default. Please refer to Variable types for possible types.

`nameprefix`

Name prefix for new variables. Optional, `"C"` by default. The actual name and the index of the variables are automatically generated by COPT.

Example

```
# Add three-dimensional integer variable, 6 variables in total
x = m.addVars(1, 2, 3, vtype=COPT.INTEGER)
# Add two continuous variable y, whose indice is designated by elements in tuplelist and␣
↪prefix is "tl"
tl = tuplelist([(0, 1), (1, 2)])
y  = m.addVars(tl, nameprefix="tl")
```

Model.addMVar()

Synopsis

`addMVars(shape, lb=0.0, ub=COPT.INFINITY, obj=0.0, vtype=COPT.CONTINUOUS, nameprefix="")`

Description

Add *MVar Class* object to the model. It is used in matrix modeling and can be operated like a multidimensional array in NumPy, its shape and dimensions are similarly defined.

Arguments

`shape`

The value is an integer, or a tuple of integers. which represents the shape of a *MVar Class* object.

`lb`

The lower bound of the variable. Optional parameter, defaults to 0.0.

`ub`

The upper bound of the variable. Optional parameter, defaults to `COPT.INFINITY`.

`obj`

The objective function coefficients for the variables. Optional parameter, defaults to 0.0.

`vtype`

The type of the variable. Optional parameter, the default is `COPT.CONTINUOUS`, see the possible values :ref: *Variable type <chap-PyAPI_Const_VarType>*.

`nameprefix`

Variable name prefix. Optional parameter, the default is "", its actual name is automatically generated by combining the subscript of the variable.

Return value

Returns a *MVar Class* object

Example

```
model.addMVar((2, 3), lb=0.0, nameprefix="mx")
```

Model.addConstr()

Synopsis

```
addConstr(lhs, sense=None, rhs=None, name="")
```

Description

Add a linear constraint, a positive semi-definite constraint or an indicator constraint to a model, and return the added *Constraint Class* object, *PsdConstraint Class* object or *GenConstr Class* object.

If a linear constraint added, then the parameter `lhs` can take the value of *Var Class* object, *LinExpr Class* object or *ConstrBuilder Class* object; If a positive semi-definite constraint added, then the parameter `lhs` can take the value of *PsdExpr Class* object, or *PsdConstrBuilder Class* object; If an indicator constraint added, then the parameter `lhs` is *GenConstrBuilder Class* object, ignoring other parameters.

Arguments

`lhs`

Left-hand side expression for new linear constraint or constraint builder.

`sense`

Sense for the new constraint. Optional, None by default. Please refer to *Constraint type* for possible values.

`rhs`

Right-hand side expression for the new constraint. Optional, None by default. It can be a constant, or *Var Class* object, or *LinExpr Class* object.

`name`

Name for new constraint. Optional, "" by default, generated by solver automatically.

Example

```
# Add a linear constraint: x + y == 2
m.addConstr(x + y, COPT.EQUAL, 2)
# Add a linear constraint: x + 2*y >= 3
m.addConstr(x + 2*y >= 3.0)
# Add an indicator constraint
m.addConstr((x == 1) >> (2*y + 3*z <= 4))
```

Model.addBoundConstr()

Synopsis

```
addBoundConstr(expr, lb=-COPT.INFINITY, ub=COPT.INFINITY, name="")
```

Description

Add a constraint with a lower bound and an upper bound to a model and return the added *Constraint Class* object.

Arguments

`expr`

Expression for the new constraint, which can be *Var Class* object or *LinExpr Class* object.

`lb`

Lower bound for the new constraint. Optional, `-COPT.INFINITY` by default.

`ub`

Upper bound for the new constraint. Optional, `COPT.INFINITY` by default.

`name`

Name for new constraint. Optional, `""` by default, automatically generated by solver.

Example

```
# Add linear bilateral constraint: -1 <= x + y <= 1
m.addBoundConstr(x + y, -1.0, 1.0)
```

Model.addConstrs()

Synopsis

```
addConstrs(generator, nameprefix="R")
```

Description

Add a set of linear constraints to a model.

If paramter `generator` is integer, the return a *ConstrArray Class* object consisting of `generator` number of empty *Constraint Class* objects, and users need to specify these constraints.

If parameter `generator` is expression generator, then return a *tupledict Class* object whose key is the indice of linear constraint and value is the corresponding *Constraint Class* object. Every iteration generates a *Constraint Class* object.

If the parameter `generator` is a matrix expression generator, return a *MConstr Class* object.

Arguments

`generator`

A generator expression, where each iteration produces a *Constraint Class* object, or a matrix expression builder.

`nameprefix`

Name prefix for new constraints. Optional, "R" by default. The actual name and the index of the constraints are automatically generated by COPT.

Example

```
# Add 10 linear constraints, each constraint shaped like: x[0] + y[0] >= 2.0
m.addConstrs(x[i] + y[i] >= 2.0 for i in range(10))
```

Model.addMConstr()

Synopsis

```
addMConstr(A, x, sense, b, nameprefix="")
```

Description

By means of matrix modeling, a set of linear constraints are added to the model. If the value of `sense` here is `COPT.LESS_EQUAL` , the added constraint is $Ax <= b$.

It is more convenient to generate *MLinExpr Class* objects by matrix multiplication, and then use overloaded comparison operators to generate *MConstrBuilder Class* object, which can be used as input of `Model.addConstrs()` to generate a set of linear constraints.

Arguments

`A`

Parameter A is a two-dimensional NumPy matrix, SciPy compressed sparse column matrix (`csc_matrix`) or compressed sparse row matrix (`csr_matrix`).

`x`

The variable corresponding to the linear term can be a *MVar Class* object, a set of *Var Class* objects. If it is empty, but the parameter c is not empty, all variables in the model are taken.

`sense`

The type of constraint. Possible values refer to *Constraint type*.

`b`

The value on the right side of the constraint, usually a floating point number, can also be a set of numbers, or a one-dimensional array of NumPy.

`nameprefix`

Constraint name prefix.

Return value

Returns a *MConstr Class* object

Example

```
A = np.full((2, 3), 1)
mx = model.addMVar(3, nameprefix="mx")
mc = model.addMConstr(A, mx, 'L', 1.0, nameprefix="mc")
```

Model.addSOS()

Synopsis

```
addSOS(sostype, vars, weights=None)
```

Description

Add a SOS constraint to model and return the added *SOS Class* object.

If param `sostype` is *SOSBuilder Class* object, then the values of param `vars` and param``weights`` will be ignored;

If param `sostype` is SOS constraint type, it can be valued as SOS-constraint types, then param `vars` represents variables of SOS constraint, taking the value of *VarArray Class* object, list, dict or *tupledict Class* object;

If param `weights` is `None`, then variable weights of SOS constraints will be automatically generated by solver. Otherwise take the input from user as weights, possible values can be list, dictionary or *tupledict Class* object.

Arguments

`sostype`

SOS contraint type or SOS constraint builder.

`vars`

Variables of SOS constraints.

`weights`

Weights of variables in SOS constraints, optional, `None` by default.

Example

```
# Add an SOS1 constraint, including variable x and y, weights of 1 and 2.
m.addSOS(COPT.SOS_TYPE1, [x, y], [1, 2])
```

Model.addGenConstrIndicator()

Synopsis

```
addGenConstrIndicator(binvar, binval, lhs, sense=None, rhs=None)
```

Description

Add an Indicator constraint to a model and return the added *GenConstr Class* object.

If the parameter `lhs` is *ConstrBuilder Class* object, then the values of parameter `sense` and parameter `rhs` will be ignored.

If parameter `lhs` represents Left-hand side expression, it can take value of *Var Class* object or *LinExpr Class* object.

Arguments

`binvar`

Indicator variable.

`binval`

Value of indicator variable, can be `True` or `False`.

`lhs`

Left-hand side expression for the linear constraint triggered by the indicator or linear constraint builder.

`sense`

Sense for the linear constraint. Optional, `None` by default. Please refer to *Constraint type* for possible values.

`rhs`

Right-hand-side value for the linear constraint triggered by the indicator. Optional, `None` by default, value type is constant.

Example

```
# Add an indicator constraint, if x is True, then the linear constraint y + 2*z >= 3 should␣
↪hold.
m.addGenConstrIndicator(x, True, y + 2*z >= 3)
```

Model.addGenConstrMin()

Synopsis

`addGenConstrMin(resvar, vars, constant=None, name="")`

Description

Add a constraint of the form $y = \min\{x_1, x_2, \cdots, x_n, c\}$ to the model.

Arguments

`resvar`

The term `y` on the left side of the equation and can be an object of class `Var`.

`vars`

The variable of the min{} function on the right side of the equation,

Possible values are `list` class objects.

`constant`

The constant term in the min{} function on the right side of the equation,

Optional parameter, the possible value is a floating number,

The default value is `None`.

`name`

Constraint name, optional parameter, default value is `""` .

Return value

It returns a `GenConstrX` Class object.

Model.addGenConstrMax()

Synopsis

```
addGenConstrMax(resvar, vars, constant=None, name="")
```

Description

Add a constraint of the form $y = \max\{x_1, x_2, \cdots, x_n, c\}$ to the model.

Arguments

`resvar`

The term `y` on the left side of the equation and can be an object of class `Var` .

`vars`

The variable of the max{} function on the right side of the equation.

Possible values are `list` class objects.

`constant`

The constant term in the max{} function on the right side of the equation.

Optional parameter, the possible value is a floating number.

The default value is `None` .

`name`

Constraint name, optional parameter, default value is `""` .

Return value

It returns a `GenConstrX` Class object.

Model.addGenConstrAbs()

Synopsis

```
addGenConstrAbs(resvar, argvar, name="")
```

Description

Add a constraint of the form $y = |x|$ to the model.

Arguments

`resvar`

y , possible values are objects of class `Var` or class `LinExpr`.

`argvar`

x , the possible value is object of class `Var` .

`name`

Constraint name, optional parameter, default value is `""` .

Return value

It returns a `GenConstrX` Class object.

Model.addGenConstrAnd()

Synopsis

```
addGenConstrAnd(resvar, vars, name="")
```

Description

Add a logical `and` constraint of the form $y = x_1$ and $x_2 \cdots$ and x_n to the model.

Arguments

`resvar`

The term `y` on the left side of the equation and can be an object of class `Var` .

`vars`

Elements connected by logical operator `and` x_i, for $i \in \{1, 2, \cdots, n\}$

Possible values are `List` class (where the elements are binary `Var` class objects).

`name`

Constraint name, optional parameter, default value is `""` .

Return value

It returns a `GenConstrX` Class object.

Model.addGenConstrOr()

Synopsis

```
addGenConstrOr(resvar, vars, name="")
```

Description

Add a logical `or` constraint of the form $y = x_1$ or $x_2 \cdots$ or x_n to the model.

Arguments

`resvar`

The term `y` on the left side of the equation and can be an object of class `Var` .

`vars`

Elements connected by logical operator `or` x_i, for $i \in \{1, 2, \cdots, n\}$

Possible values are `List` class (where the elements are binary `Var` class objects).

`name`

Constraint name, optional parameter, default value is `""` .

Return value

It returns a `GenConstrX` Class object.

Model.addGenConstrPWL()

Synopsis

```
addGenConstrPWL(xvar, yvar, xpts, ypts, name="")
```

Description

Add a constraint of the form $y = f(x)$, where a piecewise linear function is defined as:

$$f(v) = \begin{cases} \tilde{y}_1 + \frac{\tilde{y}_2 - \tilde{y}_1}{\tilde{x}_2 - \tilde{x}_1}(v - \tilde{x}_1), & \text{if } v \le x_1 \\ \tilde{y}_i + \frac{\tilde{y}_{i+1} - \tilde{y}_i}{\tilde{x}_{i+1} - \tilde{x}_i}(v - \tilde{x}_i), & \text{if } \tilde{x}_i \le v \le \tilde{x}_{i+1} \\ \tilde{y}_n + \frac{\tilde{y}_n - \tilde{y}_{n-1}}{\tilde{x}_n - \tilde{x}_{n-1}}(v - \tilde{x}_n), & \text{if } v \ge \tilde{x}_n \end{cases}$$

Arguments

`xvar`

`x`, which can be an object of class `Var`.

`yvar`

The term `y` on the left side of the equation,

Possible values are objects of class `Var` or class `LinExpr`.

`xpts`

$\tilde{\boldsymbol{x}}$, the abscissa of the segmentation point.

It should be arranged in ascending order of values, possible values are `List` class.

`ypts`

$\tilde{\boldsymbol{y}}$, the vertical coordinate of the segmentation point,

Possible values are `List` class.

`name`

Constraint name, optional parameter, default value is `""` .

Return value

It returns a `GenConstrX` Class object.

Model.addConeByDim()

Synopsis

```
addConeByDim(dim, ctype, vtype, nameprefix="ConeV")
```

Description

Add a Second-Order-Cone (SOC) constraint with given dimension, and return the added *Cone Class* object.

Arguments

`dim`

Dimension of SOC constraint.

`ctype`

Type of SOC constraint.

`vtype`

Variable types of SOC constraint.

`nameprefix`

Name prefix of variables in SOC constraint. Optional, default to `"ConeV"`.

Example

```
# Add a 5 dimension rotated SOC constraint
m.addConeByDim(5, COPT.CONE_RQUAD, None)
```

Model.addCone()

Synopsis

```
addCone(vars, ctype)
```

Description

Add a Second-Order-Cone (SOC) constraint with given variables.

If argument `vars` is a *ConeBuilder Class* object, then the value of argument `ctype` will be ignored; If argument `vars` are variables, the optional values are *VarArray Class* objects, Python list, Python dictionary or *tupledict Class* objects, argument `ctype` is the type of SOC constraint.

Arguments

`vars`

Variables of SOC constraint.

`ctype`

Type of SOC constraint.

Example

```
# Add a SOC constraint with [z, x, y] as variables
m.addCone([z, x, y], COPT.CONE_QUAD)
```

Model.addQConstr()

Synopsis

```
addQConstr(lhs, sense=None, rhs=None, name="")
```

Description

Add a linear or quadratic constraint, and return the added *Constraint Class* object or *QConstraint Class* object.

If the constraint is linear, then value of parameter `lhs` can be taken *Var Class* object, *LinExpr Class* object or *ConstrBuilder Class* object; If the constraint is quadratic, then value of parameter `lhs` can be taken *QConstrBuilder Class* object, or *MQConstrBuilder Class* object and other parameters will be ignored.

Arguments

`lhs`

Left-hand side expression for new constraint or constraint builder.

`sense`

Sense for the new constraint. Optional, None by default. Please refer to *Constraint type* for possible values.

`rhs`

Right-hand side expression for the new constraint. Optional, None by default. It can be a constant, *Var Class* object, *LinExpr Class* object or *QuadExpr Class* object.

`name`

Name for new constraint. Optional, `""` by default, generated by solver automatically.

Example

```
# add a linear equality: x + y == 2
m.addQConstr(x + y, COPT.EQUAL, 2)
# add a quadratic inequality: x*x + y*y <= 3
m.addQConstr(x*x + y*y <= 3.0)
```

Model.addMQConstr()

Synopsis

```
addMQConstr(Q, c, sense, rhs, xQ_L=None, xQ_R=None, xc=None,
name="")
```

Description

By means of matrix modeling, a quadratic constraint is added to the model. If the value of `sense` here is `COPT.LESS_EQUAL` , the added constraint is $x_{Q_L} Q x_{Q_R} + cx_c <= rhs$.

It is more convenient to generate *MQuadExpr Class* objects by matrix multiplication, and then use overloaded comparison operators to generate *MQConstrBuilder Class* object, which can be used as input of `Model.addQConstr()` to generate quadratic constraints.

Arguments

`Q`

If the quadratic term is not empty, the parameter Q needs to be provided, which is a two-dimensional NumPy matrix, SciPy compressed sparse column matrix (`csc_matrix`) or compressed sparse row matrix (`csr_matrix`).

`c`

If the item is non-empty, you need to provide the parameter c, which is a one-dimensional NumPy array, or a Python list.

`sense`

The type of constraint. Possible values refer to *Constraint type*.

`rhs`

The value on the right side of the constraint, usually a floating point number.

`xQ_L`

The variable on the left side of the quadratic term, can be a *MVar Class* object, an array of *Var Class* objects. If empty, all variables in the model are taken.

`xQ_R`

The variable on the right side of the quadratic term, can be a *MVar Class* object, a set of *Var Class* objects. If empty, all variables in the model are taken.

`xc`

The variable corresponding to the linear term can be a *MVar Class* object, a set of *Var Class* objects. If it is empty, but the parameter c is not empty, all variables in the model are taken.

`name`

constraint name.

Return value

Returns a *QConstraint Class* object

Example

```
Q = np.full((3, 3), 1)
mx = model.addMVar(3, nameprefix="mx")
mqc = model.addMQConstr(Q, None, 'L', 1.0, mx, mx, None, name="mqc")
```

Model.addPsdVar()

Synopsis

```
addPsdVar(dim, name="")
```

Description

Add a positive semi-definite variable.

Arguments

`dim`

Dimension for the positive semi-definite variable.

`name`

Name for the positive semi-definite variable.

Example

```
# Add a three-dimensional positive semi-definite variable, "X"
m.addPsdVar(3, "X")
```

Model.addPsdVars()

Synopsis

```
addPsdVars(dims, nameprefix="PSDV")
```

Description

Add multiple new positive semi-definite variables to a model.

Arguments

`dim`

Dimensions for new positive semi-definite variables.

`nameprefix`

Name prefix for new positive semi-definite variables.

Example

```
# Add two three-dimensional positive semi-definite variables
m.addPsdVars([3, 3])
```

Model.addUserCut()

Synopsis

```
addUserCut(lhs, sense = None, rhs = None, name="")
```

Description

Add a user cut to the MIP model.

Arguments

`lhs`

Left-hand side expression for the new user cut. It can take the value of *Var Class* object, *LinExpr Class* object or *ConstrBuilder Class* object.

`sense`

The sense of the new user cut. Please refer to *Constraint type* for possible values.

Optional. None by default.

`rhs`

Right-hand side expression for the new user cut.

Optional. None by default.

It can be a constant, or *Var Class* object, or *LinExpr Class* object.

`name`

Name for the new user cut. Optional, `""` by default, automatically generated by solver.

Example

```
model.addUserCut(x+y <= 1)

model.addUserCut(x+y == [0, 1])
```

Model.addUserCuts()

Synopsis

```
addUserCuts(generator, nameprefix="U")
```

Description

Add a set of user cuts to the MIP model.

Arguments

`generator`

A generator expression, where each iteration produces a *Constraint Class* object.

`nameprefix`

Name prefix for new user cuts. Optional, `"U"` by default. The actual name and the index of the constraints are automatically generated by COPT.

Example

```
model.addUserCuts(x[i]+y[i] <= 1 for i in range(10))
```

Model.addLazyConstr()

Synopsis

```
addLazyConstr(lhs, sense = None, rhs = None, name="")
```

Description

Add a lazy constraint to the MIP model.

Arguments

`lhs`

Left-hand side expression for the new lazy constraint. It can take the value of *Var Class* object, *LinExpr Class* object or *ConstrBuilder Class* object.

`sense`

The sense of the lazy constraint. Please refer to *Constraint type* for possible values.

Optional. None by default.

`rhs`

Right-hand side expression for the new lazy constraint.

Optional. None by default.

It can be a constant, or *Var Class* object, or *LinExpr Class* object.

`name`

Name for the new lazy constraint. Optional, `""` by default, automatically generated by solver.

Example

```
model.addLazyConstr(x+y <= 1)
model.addLazyConstr(x+y == [0, 1])
```

Model.addLazyConstrs()

Synopsis

```
addLazyConstrs(generator, nameprefix="L")
```

Description

Add a set of lazy constraints to the MIP model.

Arguments

`generator`

A generator expression, where each iteration produces a *Constraint Class* object.

`nameprefix`

Name prefix for new lazy constraints. Optional, `"L"` by default. The actual name and the index of the constraints are automatically generated by COPT.

Example

```
model.addLazyConstrs(x[i]+y[i] <= 1 for i in range(10))
```

Model.addSparseMat()

Synopsis

```
addSparseMat(dim, rows, cols=None, vals=None)
```

Description

Add a sparse symmetric matrix in triplet format

Arguments

`dim`

Dimension for the matrix.

`rows`

Row indices for accessing rows of non-zero elements.

`cols`

Column indices for accessing columns of non-zero elements.

`vals`

Coefficient values for non-zero elements

Example

```
# Add a tree-dimentional symmetric matrix
m.addSparseMat(3, [0, 1, 2], [0, 1, 2], [2.0, 5.0, 8.0])
```

Model.addDenseMat()

Synopsis

```
addDenseMat(dim, vals)
```

Description

Add a dense symmetric matrix

Arguments

`dim`

Dimension for the matrix.

`vals`

Coefficient values, which can be a constant or a list.

Example

```
# Add a tree-dimentional matrix (filled with ones)
m.addDenseMat(3, 1.0)
```

Model.addDiagMat()

Synopsis

```
addDiagMat(dim, vals, offset=None)
```

Description

Add a diagnal symmetric matrix

Arguments

`dim`

Dimension for the matrix.

`vals`

Coefficient values, which can be a constant or a list.

`offset`

Offset of diagnal elements. Positive: above the diagnal; Negative: below the diagnal

Example

```
# Add a tree-dimentional identity matrix
m.addDiagMat(3, 1.0)
```

Model.addOnesMat()

Synopsis

```
addOnesMat(dim)
```

Description

Add a matrix filled with ones.

Arguments

`dim`

Dimension for the matrix.

Example

```
# Add a tree-dimentional matrix (filled with ones)
m.addOnesMat(3)
```

Model.addEyeMat()

Synopsis

```
addEyeMat(dim)
```

Description

Add an identity matrix

Arguments

`dim`

Dimension for the matrix.

Example

```
# Add a tree-dimentional identity matrix
m.addEyeMat(3)
```

Model.setObjective()

Synopsis

```
setObjective(expr, sense=None)
```

Description

Set the model objective.

Arguments

`expr`

Objective expression. Argument can be a constant, *Var Class* object, *LinExpr Class* object, *QuadExpr Class* object, *MLinExpr Class* object, or *MQuadExpr Class* object.

`sense`

Optimization sense. Optional, `None` by default, which means no change to objective sense. Users can get access to current objective sense by attribute `ObjSense` . Please refer to *Optimization directions* for possible values.

Example

```
# Set objective function = x+y, optimization sense is maximization.
m.setObjective(x + y, COPT.MAXIMIZE)
```

Model.setMObjective()

Synopsis

```
setMObjective(Q, c, constant, xQ_L=None, xQ_R=None, xc=None,
sense=None)
```

Description

Set the secondary objective of the model by matrix modeling. Objective functions of the form $x_{Q_L} Q x_{Q_R} + c x_c + constant$ can be added.

Even more convenient is to generate a *MQuadExpr Class* object via matrix multiplication, available as the input to `setObjective()` to set the objective function.

Arguments

`Q`

If the quadratic term is not empty, the parameter Q needs to be provided, which is a two-dimensional NumPy matrix, SciPy compressed sparse column matrix (`csc_matrix`) or compressed sparse row matrix (`csr_matrix`).

`c`

If the item is non-empty, you need to provide the parameter c, which is a one-dimensional NumPy array, or a Python list.

`constant`

Constant term, usually a floating point number.

`xQ_L`

The variable on the left side of the quadratic term, can be a *MVar Class* object, an array of *Var Class* objects. If empty, all variables in the model are taken.

`xQ_R`

The variable on the right side of the quadratic term, can be a *MVar Class* object, a set of *Var Class* objects. If empty, all variables in the model are taken.

`xc`

The variable corresponding to the linear term can be a *MVar Class* object, a set of *Var Class* objects. If it is empty, but the parameter c is not empty, all variables in the model are taken.

`sense`

The optimization direction of the objective function. Optional parameter, the default is `None`, which means that the optimization direction of the model will not be changed. The current optimization direction of the model is viewed through the property `ObjSense`. See *Optimization Direction* for possible values.

Example

```
Q = np.full((3, 3), 1)
mx = model.addMVar(3, nameprefix="mx")
mqc = model.setMObjective(Q, None, 0.0, mx, mx, None, sense=COPT.MINIMIZE)
```

Model.setObjSense()

Synopsis

`setObjSense(sense)`

Description

Set optimization sense.

Arguments

`sense`

Optimization sense. Please refer to *Optimization directions* for possible values.

Example

```
# Set optimization sense as maximization
m.setObjSense(COPT.MAXIMIZE)
```

Model.setObjConst()

Synopsis

`setObjConst(const)`

Description

Set constant objective offset.

Arguments

`const`

Constant objective offset.

Example

```
# Set constant objective offset 1.0
m.setObjConst(1.0)
```

Model.getObjective()

Synopsis

```
getObjective()
```

Description

Retrieve current model objective. Return a *LinExpr Class* object.

Example

```
# Retrieve the optimization objective.
obj = m.getObjective()
```

Model.delQuadObj()

Synopsis

```
delQuadObj()
```

Description

Deletes the quadratic terms from the quadratic objective function.

Example

```
# Deletes the quadratic terms from the quadratic objective function
m.delQuadObj()
```

Model.delPsdObj()

Synopsis

```
delPsdObj()
```

Description

Delete the positive semi-definite terms from the objective function

Example

```
# Delete the positive semi-definite terms from the objective function
m.delPsdObj()
```

Model.getCol()

Synopsis

```
getCol(var)
```

Description

Retrieve the list of constraints in which a variable participates, Return value is a *Column Class* object that captures the set of constraints in which the variable participates.

Example

```
# Get column that captures the set of constraints in which x participates.
col = m.getCol(x)
```

Model.getRow()

Synopsis

```
getRow(constr)
```

Description Retrieve the list of variables that participate in a specific constraint and return a *LinExpr Class* object.

Example

```
# Return variables that participate in conx.
linexpr = m.getRow(conx)
```

Model.getQuadRow()

Synopsis

```
getQuadRow(qconstr)
```

Description

Retrieve the list of variables that participate in a specific quadratic constraint and return a *QuadExpr Class* object.

Example

```
# Return variables that participate in qconx
quadexpr = m.getQuadRow(qconx)
```

Model.getPsdRow()

Synopsis

```
getPsdRow(constr)
```

Description

Retrieve the list of variables that participate in a specific positive semi-definite constraint and return a *PsdExpr Class* object.

Example

```
# Retrieve the row corresponding to a specific positive semi-definite constraint
psdexpr = m.getPsdRow(psdcon)
```

Model.getVar()

Synopsis

```
getVar(idx)
```

Description

Retrieve a variable according to its index in the coefficient matrix. Return a *Var Class* object.

Arguments

idx

Index of the desired variable in the coefficient matrix, starting with 0.

Example

```
# Retrive variable with indice of 1.
 x = m.getVar(1)
```

Model.getVarByName()

Synopsis

getVarByName(name)

Description

Retrieves a variable by name. Return a *Var Class* object.

Arguments

name

Name of the desired variable.

Example

```
#  Retrive variable with name "x".
x = m.getVarByName("x")
```

Model.getVars()

Synopsis

getVars()

Description

Retrieve all variables in the model. Return a *VarArray Class* object.

Example

```
# Retrieve all variables in the model
vars = m.getVars()
```

Model.getConstr()

Synopsis

getConstr(idx)

Description

Retrieve a constraint by its indice in the coefficient matrix. Return a *Constraint Class* object.

Arguments

idx

Index of the desired constraint in the coefficient matrix, starting with 0.

Example

```
# Retrive linear constraint with indice of 1.
r = m.getConstr(1)
```

Model.getConstrByName()

Synopsis

`getConstrByName(name)`

Description

Retrieves a linear constraint by name. Return a *Constraint Class* object.

Arguments

`name`

The name of the constraint.

Example

```
# Retrieve linear constraint with name "r".
r = m.getConstrByName("r")
```

Model.getConstrs()

Synopsis

`getConstrs()`

Description

Retrieve all constraints in the model. Return a *ConstrArray Class* object.

Example

```
# Retrieve all constraints in the model
cons = m.getConstrs()
```

Model.getConstrBuilders()

Synopsis

`getConstrBuilders(constrs=None)`

Description

Retrieve linear constraint builders in current model.

If parameter `constrs` is `None`, then return a *ConstrBuilderArray Class* object composed of all linear constraint builders.

If parameter `constrs` is *Constraint Class* object, then return the *ConstrBuilder Class* object corresponding to the specific constraint.

If parameter `constrs` is a list or a *ConstrArray Class* object, then return a *ConstrBuilderArray Class* object composed of specified constraints' builders.

If parameter `constrs` is dictionary or *tupledict Class* object, then the indice of the specified constraint is returned as key, the value is a *tupledict Class* object composed of the specified constraints' builders.

Arguments

`constrs`

The specified linear constraint. Optional, `None` by default.

Example

```
# Retrieve all of linear constraint builders.
conbuilders = m.getConstrBuilders()
# Retrive the builder corresponding to linear contstraint x.
conbuilders = m.getConstrBuilders(x)
# Retrieve builders corresponding to linear constraint x and y.
conbuilders = m.getConstrBuilders([x, y])
# Retrieve builders corresponding to linear constraint in tupledict object xx.
conbuilders = m.getConstrBuilders(xx)
```

Model.getSOS(sos)

Synopsis

`getSOS(sos)`

Description

Retrieve the SOS constraint builder corresponding to specific SOS constraint. Return a *SOSBuilder Class* object

Arguments

`sos`

The specified SOS constraint.

Example

```
# Retrieve the builder corresponding to SOS constraint sosx.
sosbuilder = m.getSOS(sosx)
```

Model.getSOSs()

Synopsis

`getSOSs()`

Description

Retrieve all SOS constraints in model and return a *SOSArray Class* object.

Example

```
#  Retrieve all SOS constraints in model.
soss = m.getSOSs()
```

Model.getSOSBuilders()

Synopsis

`getSOSBuilders(soss=None)`

Description

Retrieve the SOS constraint builder corresponding to the specified SOS constraint.

If parameter `soss` is `None`, then return a *SOSBuilderArray Class* object consisting of builders corresponding to all SOS constraints.

If parameter `soss` is *SOS Class* object, then return a *SOSBuilder Class* corresponding to the specified SOS constraint.

If parameter `soss` is list or *SOSArray Class* object, then return a *SOSBuilderArray Class* object consisting of builders corresponding to the specific SOS constraints.

Arguments

`soss` The specific SOS constraint. Optional, `None` by default.

Example

```
# Retrieve builders corresponding to all SOS constraints in the model.
soss = m.getSOSBuilders()
```

Model.getGenConstrIndicator()

Synopsis

`getGenConstrIndicator(genconstr)`

Description

Retrieve the builder corresponding to specific Indicator constraint. Return a *GenConstrBuilder Class* object.

Arguments

`genconstr`

The specified Indicator constraint.

Example

```
# Retrieve the builder corresponding to Indicator constraint genx.
indic = m.getGenConstrIndicator(genx)
```

Model.getCones()

Synopsis

`getCones()`

Description

Retrieve all Second-Order-Cone (SOC) constraints in model, and return a *ConeArray Class* object.

Example

```
# Retrieve all SOC constraints
cones = m.getCones()
```

Model.getConeBuilders()

Synopsis

`getConeBuilders(cones=None)`

Description

Retrieve Second-Order-Cone (SOC) constraint builders for given SOC constraints.

If argument `cones` is `None`, then return a *ConeBuilderArray Class* object consists of all SOC constraints' builders; If argument `cones` is *Cone Class* object, then return a *ConeBuilder Class* object of given SOC constraint; If `cones` is Python list or *ConeArray Class* object, then return a *ConeBuilderArray Class* object consists of builders of given SOC constraints.

Arguments

`cones`

Given SOC constraints. Optional, default to `None`.

Example

```
# Retrieve all SOC constraints' builders
cones = m.getConeBuilders()
```

Model.getQConstr()

Synopsis

`getQConstr(idx)`

Description

Retrieve a quadratic constraint by its indice, and return a *QConstraint Class* object.

Arguments

`idx`

Index of the desired quadratic constraint, starting with `0`.

Example

```
# Retrieve a quadratic constraint with indice of 1
qr = m.getQConstr(1)
```

Model.getQConstrByName()

Synopsis

`getQConstrByName(name)`

Description

Retrieve a quadratic constraint by its name, and return a *QConstraint Class* object.

Arguments

`name`

Name of the desired quadratic constraint.

Example

```
# Retrieve a quadratic constraint with name "qr"
qr = m.getQConstrByName("qr")
```

Model.getQConstrs()

Synopsis

`getQConstrs()`

Description

Retrieve all quadratic constraints in the model. Return a *QConstrArray Class* object.

Example

```
# Retrieve all quadratic constraints in the model
qcons = m.getQConstrs()
```

Model.getQConstrBuilders()

Synopsis

```
getQConstrBuilders(qconstrs=None)
```

Description

Retrieve quadratic constraint builders in current model.

If parameter `qconstrs` is `None`, then return a *QConstrBuilderArray Class* object composed of all quadratic constraint builders.

If parameter `qconstrs` is *QConstraint Class* object, then return the *QConstrBuilder Class* object corresponding to the specific quadratic constraint.

If parameter `qconstrs` is a list or a *QConstrArray Class* object, then return a *QConstrBuilderArray Class* object composed of specified quadratic constraints' builders.

If parameter `qconstrs` is dictionary or *tupledict Class* object, then the indice of the specified quadratic constraint is returned as key, the value is a *tupledict Class* object composed of the specified quadratic constraints' builders.

Arguments

`qconstrs`

The specified quadratic constraint. Optional, `None` by default.

Example

```
# Retrieve all of quadratic constraint builders.
qconbuilders = m.getQConstrBuilders()
# Retrive the builder corresponding to quadratic contstraint qx.
qconbuilders = m.getQConstrBuilders(qx)
# Retrieve builders corresponding to quadratic constraint qx and qy.
qconbuilders = m.getQConstrBuilders([qx, qy])
# Retrieve builders corresponding to quadratic constraint in tupledict object qxx.
qconbuilders = m.getQConstrBuilders(qxx)
```

Model.getPsdVar()

Synopsis

```
getPsdVar(idx)
```

Description

Retrieve a positive semi-definite variable according to its index in the model. Return a *PsdVar Class* object.

Arguments

`idx`

Index of the desired positive semi-definite variable in the model, starting with 0.

Example

```
# Retrieve a positive semi-definite variable with index of 1
x = m.getPsdVar(1)
```

Model.getPsdVarByName()

Synopsis

```
getPsdVarByName(name)
```

Description

Retrieve a positive semi-definite variable by name. Return a *PsdVar Class* object.

Arguments

`name`

The name of the positive semi-definite variable.

Example

```
# Retrieve a positive semi-definite variable with name "x".
x = m.getPsdVarByName("x")
```

Model.getPsdVars()

Synopsis

```
getPsdVars()
```

Description

Retrieve all positive semi-definite variables in the model, and return a *PsdVarArray Class* object.

Example

```
# Retrieve all positive semi-definite variables in the model.
vars = m.getPsdVars()
```

Model.getPsdConstr()

Synopsis

```
getPsdConstr(idx)
```

Description

Retrieve the positive semi-definite constraint according to its index in the model. Return a *PsdConstraint Class* object.

Arguments

`idx`

Index for the positive semi-definite constraint, starting with 0.

Example

```
# Retrieve the positive semi-definite constraint with index of 1
r = m.getPsdConstr(1)
```

Model.getPsdConstrByName()

Synopsis

```
getPsdConstrByName(name)
```

Description

Retrieve a positive semi-definite constraint by name. Return a *PsdConstraint Class* object.

Arguments

`name`

The name of the positive semi-definite constraint.

Example

```
# Retrieve the positive semi-definite constraint with name "r".
r = m.getPsdConstrByName("r")
```

Model.getPsdConstrs()

Synopsis

```
getPsdConstrs()
```

Description

Retrieve all positive semi-definite constraints in the model. Return a *PsdConstrArray Class* object.

Example

```
# Retrieve all positive semi-definite constraints in the model
cons = m.getPsdConstrs()
```

Model.getPsdConstrBuilders()

Synopsis

```
getPsdConstrBuilders(constrs=None)
```

Description

Retrieve positive semi-definite constraint builders in current model.

If parameter `constrs` is `None`, then return a *PsdConstrBuilderArray Class* object composed of all positive semi-definite constraint builders.

If parameter `constrs` is *PsdConstraint Class* object, then return the *PsdConstrBuilder Class* object corresponding to the specific positive semi-definite constraint.

If parameter `constrs` is a list or a *PsdConstrArray Class* object, then return a *PsdConstrBuilderArray Class* object composed of specified positive semi-definite constraints' builders.

If parameter `constrs` is dictionary or *tupledict Class* object, then the indice of the specified positive semi-definite constraint is returned as key, the value is a *tupledict Class* object composed of the specified positive semi-definite constraints' builders.

Arguments

`constrs`

The specified positive semi-definite constraint. Optional, `None` by default.

Example

```
# Retrieve all of positive semi-definite constraint builders.
conbuilders = m.getPsdConstrBuilders()
# Retrive the builder corresponding to positive semi-definite contstraint x.
conbuilders = m.getPsdConstrBuilders(x)
# Retrieve builders corresponding to positive semi-definite constraint x and y.
conbuilders = m.getPsdConstrBuilders([x, y])
# Retrieve builders corresponding to positive semi-definite constraint in tupledict object xx.
conbuilders = m.getPsdConstrBuilders(xx)
```

Model.getCoeff()

Synopsis

```
getCoeff(constr, var)
```

Description

Get the coefficient of variable in linear constraint.

Arguments

`constr`

The requested linear constraint.

`var`

The requested variable.

Example

```
# Get the coefficient of variable x in linear constraint c
coeff = m.getCoeff(c, x)
```

Model.setCoeff()

Synopsis

```
setCoeff(constr, var, newval)
```

Description

Set the coefficient of variable in linear constraint.

Arguments

`constr`

The requested linear constraint.

`var`

The requested variable.

`newval`

New coefficient.

Example

```
# Set the coefficient of variable x in linear constraint c to 1.0
m.setCoeff(c, x, 1.0)
```

Model.getA()

Synopsis

```
getA()
```

Description

Get the coefficient matrix of model, returns a `scipy.sparse.csc_matrix` object. This method requires the `scipy` package.

Example

```
# Get the coefficient matrix
A = model.getA()
```

Model.loadMatrix()

Synopsis

```
loadMatrix(c, A, lhs, rhs, lb, ub, vtype=None)
```

Description

Load matrix and vector data to build model. This method requires the `scipy` package.

Arguments

`c`

Objective costs. If `None`, the objective costs are all zeros.

`A`

Coefficient matrix. Must be of type `scipy.sparse.csc_matrix`.

`lhs`

Lower bounds of constraints.

`rhs`

Upper bounds of constraints.

`lb`

Lower bounds of variables. If `None`, the lower bounds are all zeros.

`ub`

Upper bounds of variables. If `None`, the upper bounds are all `COPT.INFINITY`.

`vtype`

Variable types. Default to `None`, which means all variables are continuous.

Example

```
# Build model by problem matrix
m.loadMatrix(c, A, lhs, rhs, lb, ub)
```

Model.getLpSolution()

Synopsis

```
getLpSolution()
```

Description

Retrieve the values of variables, slack variables, dual variables and reduced cost of variables. Return a quad tuple object, in which each element is a list.

Example

```
# Retrieve solutions of linear model.
values, slacks, duals, redcosts = m.getLpSolution()
```

Model.setLpSolution()

Synopsis

```
setLpSolution(values, slack, duals, redcost)
```

Description

Set LP solution.

Arguments

`values`

Solution of variables.

`slack`

Solution of slack variables.

`duals`

Solution of dual variables.

`redcost`

Reduced costs of variables.

Example

```
# Set LP solution
m.setLpSolution(values, slack, duals, redcost)
```

Model.getValues()

Synopsis

```
getValues()
```

Description

Retrieve solution values of all variables in a LP or MIP. Return a Python list.

Example

```
values = m.getValues()
```

Model.getRedcosts()

Synopsis

```
getRedcosts()
```

Description

Retrieve reduced costs of all variables in a LP. Return a list.

Example

```
# Retrieve reduced cost of all variables in model.
redcosts = m.getRedcosts()
```

Model.getSlacks()

Synopsis

```
getSlacks()
```

Description

Retrieve values of all slack variables in a LP. Return a Python list.

Example

```
# Retrieve value of all slack variables in model.
 slacks = m.getSlacks()
```

Model.getDuals()

Synopsis

```
getDuals()
```

Description

Obtain values of all dual variables in a LP. Return a Python list.

Example

```
# Retrieve value of all dual variables in model.
duals = m.getDuals()
```

Model.getVarBasis()

Synopsis

```
getVarBasis(vars=None)
```

Description

Obtain basis status of specified variables.

If parameter `vars` is `None`, then return a list object consistinf of all variables' basis status. If parameter `vars` is *Var Class* object, then return basis status of the specified variable. If parameter `vars` is list or *VarArray Class* object, then return a list object consisting of the specified variables' basis status. If parameter `vars` is dictionary or *tupledict Class* object, then return indice of the specified variable as key and *tupledict Class* object consisting of the specified variables' basis status as value.

Arguments

`vars`

The specified variables. Optional, `None` by default,

Example

```
# Retrieve all variables' basis status in model.
varbasis = m.getVarBasis()
# Retrieve basis status of variable x and y.
varbasis = m.getVarBasis([x, y])
# Retrieve basis status of tupledict object xx.
varbasis = m.getVarBasis(xx)
```

Model.getConstrBasis()

Synopsis

`getConstrBasis(constrs=None)`

Description

Obtain the basis status of linear constraints in LP.

If parameter `constrs` is `None`, then return a list object consisting of all linear constraints' basis status. If parameter `constrs` is *Constraint Class* object, then return basis status of the specified linear constraint. If parameter `constrs` is list or *ConstrArray Class* object, then return a list object consisting of the specified linear constraints' basis status. If parameter `constrs` is dictionary or *tupledict Class* object, then return the indice of the specified linear constraint as key and return *tupledict Class* object consisting of the specified linear constraints' basis status as value.

Arguments

`constrs`

The specified linear constraint. Optional, `None` by default.

Example

```
# Retrieve all linear constraints' basis status in model.
conbasis = m.getConstrBasis()
# Retrieve basis status correspoding to linear constraint r0 and r1 in model.
conbasis = m.getConstrBasis([r0, r1])
# Retrieve basis status of linear constraints in tupledict rr.
conbasis = m.getConstrBasis(rr)
```

Model.getPoolObjVal()

Synopsis

`getPoolObjVal(isol)`

Description

Obtain the `isol` -th objective value in solution pool, return a constant.

Arguments

`isol`

Index of solution.

Example

```
# Obtain the second objective value
objval = m.getPoolObjVal(2)
```

Model.getPoolSolution()

Synopsis

```
getPoolSolution(isol, vars)
```

Description

Obtain variable values in the `isol` -th solution of solution pool.

If parameter `vars` is *Var Class* object, then return values of the specified variable. If parameter `vars` is list or *VarArray Class* object, then return a list object consisting of the specified variables' values. If parameter `vars` is dictionary or *tupledict Class* object, then return indice of the specified variable as key and *tupledict Class* object consisting of the specified variables' values as value.

Arguments

`isol`

Index of solution

`vars`

The specified variables.

Example

```
# Get value of x in the second solution
xval = m.getPoolSolution(2, x)
```

Model.getVarLowerIIS()

Synopsis

```
getVarLowerIIS(vars)
```

Description

Obtain IIS status of lower bounds of variables.

If parameter `vars` is *Var Class* object, then return IIS status of lower bound of variable. If parameter `vars` is list or *VarArray Class* object, then return a list object consisting of the IIS status of lower bounds of variables. If parameter `vars` is dictionary or *tupledict Class* object, then return indice of the specified variable as key and *tupledict Class* object consisting of the IIS status of lower bounds of variables as value.

Arguments

`vars`

The specified variables.

Example

```
# Retrieve IIS status of lower bounds of variable x and y.
lowerIIS = m.getVarLowerIIS([x, y])
# Retrieve IIS status of lower bounds of variables in tupledict object xx.
lowerIIS = m.getVarLowerIIS(xx)
```

Model.getVarUpperIIS()

Synopsis

```
getVarUpperIIS(vars)
```

Description

Obtain IIS status of upper bounds of variables.

If parameter `vars` is *Var Class* object, then return IIS status of upper bound of variable. If parameter `vars` is list or *VarArray Class* object, then return a list object consisting of the IIS status of upper bounds of variables. If parameter `vars` is dictionary or *tupledict Class* object, then return indice of the specified variable as key and *tupledict Class* object consisting of the IIS status of upper bounds of variables as value.

Arguments

`vars`

The specified variables.

Example

```
# Retrieve IIS status of upper bounds of variable x and y.
upperIIS = m.getVarUpperIIS([x, y])
# Retrieve IIS status of upper bounds of variables in tupledict object xx.
upperIIS = m.getVarUpperIIS(xx)
```

Model.getConstrLowerIIS()

Synopsis

```
getConstrLowerIIS(constrs)
```

Description

Obtain the IIS status of lower bounds of constraints.

If parameter `constrs` is *Constraint Class* object, then return IIS status of lower bound of constraint. If parameter `constrs` is list or *ConstrArray Class* object, then return a list object consisting of the IIS status of lower bounds of constraints. If parameter `constrs` is dictionary or *tupledict Class* object, then return the indice of the specified linear constraint as key and return *tupledict Class* object consisting of the IIS status of lower bounds of constraints.

Arguments

`constrs`

The specified linear constraint.

Example

```
# Retrieve IIS status correspoding to lower bounds of linear constraint r0 and r1 in model.
lowerIIS = m.getConstrLowerIIS([r0, r1])
# Retrieve IIS status of lower bounds of linear constraints in tupledict rr.
lowerIIS = m.getConstrLowerIIS(rr)
```

Model.getConstrUpperIIS()

Synopsis

```
getConstrUpperIIS(constrs)
```

Description

Obtain the IIS status of upper bounds of constraints.

If parameter `constrs` is *Constraint Class* object, then return IIS status of upper bound of constraint. If parameter `constrs` is list or *ConstrArray Class* object, then return a list object consisting of the IIS status of upper bounds of constraints. If parameter `constrs` is dictionary or *tupledict Class* object, then return the indice of the specified linear constraint as key and return *tupledict Class* object consisting of the IIS status of upper bounds of constraints.

Arguments

`constrs`

The specified linear constraint.

Example

```
# Retrieve IIS status correspoding to upper bounds of linear constraint r0 and r1 in model.
upperIIS = m.getConstrUpperIIS([r0, r1])
# Retrieve IIS status of upper bounds of linear constraints in tupledict rr.
upperIIS = m.getConstrUpperIIS(rr)
```

Model.getSOSIIS()

Synopsis

```
getSOSIIS(soss)
```

Description

Obtain the IIS status of SOS constraints.

If parameter `soss` is *SOS Class* object, then return IIS status of SOS constraint. If parameter `soss` is list or *SOSArray Class* object, then return a list object consisting of the IIS status of SOS constraints. If parameter `soss` is dictionary or *tupledict Class* object, then return the indice of the specified SOS constraint as key and return *tupledict Class* object consisting of the IIS status of SOS constraints.

Arguments

`soss`

The specified SOS constraint.

Example

```
# Retrieve IIS status correspoding to SOS constraint r0 and r1 in model.
sosIIS = m.getSOSIIS([r0, r1])
# Retrieve IIS status of SOS constraints in tupledict rr.
sosIIS = m.getSOSIIS(rr)
```

Model.getIndicatorIIS()

Synopsis

```
getIndicatorIIS(genconstrs)
```

Description

Obtain the IIS status of indicator constraints.

If parameter `genconstrs` is *GenConstr Class* object, then return IIS status of indicator constraint. If parameter `genconstrs` is list or *GenConstrArray Class* object, then return a list object consisting of the IIS status of indicator constraints. If parameter `genconstrs` is dictionary or *tupledict Class* object, then return the indice of the specified indicator constraint as key and return *tupledict Class* object consisting of the IIS status of indicator constraints.

Arguments

`genconstrs`

The specified indicator constraint.

Example

```
# Retrieve IIS status correspoding to indicator constraint r0 and r1 in model.
indicatorIIS = m.getIndicatorIIS([r0, r1])
# Retrieve IIS status of indicator constraints in tupledict rr.
indicatorIIS = m.getIndicatorIIS(rr)
```

Model.getAttr()

Synopsis

```
getAttr(attrname)
```

Description

Get the value of an attribute of model. Return a constant.

Arguments

`attrname`

The specified attribute name. The full list of available attributes can be found in *Attributes* section.

Example

```
# Retrieve the constant terms of objective.
objconst = m.getAttr(COPT.Attr.ObjConst)
```

Model.getInfo()

Synopsis

```
getInfo(infoname, args)
```

Description

Retrieve specified information.

If parameter `args` is *Var Class* object or *Constraint Class* object, then return info of the specified variable or constraint.

If parameter `args` is list or *VarArray Class* object or *ConstrArray Class* object, then return a list consisting of the specified variables or constraints.

If parameter `args` is dictionary or *tupledict Class* object, then return the indice of the specified variables or constraints as key and return *tupledict Class* object consisting of info corresponding to the specified variables or constraints as value.

If parameter `args` is *MVar Class* object or *MConstr Class* object, then return a `numpy.ndarray` consisting of the specified variables or constraints.

Arguments

`infoname`

The specified information name. The full list of available attributes can be found in ref:*chapInfo* section.

`args`

Variables and constraints to get information.

Example

```
# Retrieve lower bound information of all linear constraints in model.
lb = m.getInfo(COPT.Info.LB, m.getConstrs())
# Retrieve value information of variables x and y.
sol = m.getInfo(COPT.Info.Value, [x, y])
# Retrieve the dual variable value corresponding to linear constraint in tupledict object␣
↪shipconstr.
dual = m.getInfo(COPT.Info.Dual, shipconstr)
```

Model.getParam()

Synopsis

`getParam(paramname)`

Description

Retrive the current value of the specified parameter. Return a constant.

Arguments

`paramname`

The name of the parameter to get access to. The full list of available attributes can be found in *Parameters* section.

Example

```
# Retrieve current value of time limit.
timelimit = m.getParam(COPT.Param.TimeLimit)
```

Model.getParamInfo()

Synopsis

`getParamInfo(paramname)`

Description

Retrieve information of the specified optimization parameter. Return a tuple object, consisiting of param name, current value, default value, minimum value, maximum value.

Arguments

`paramname`

Name of the specified parameter. The full list of available values can be found in *Parameters* section.

Example

```
# Retrieve information of time limit.
pname, pcur, pdef, pmin, pmax = m.getParamInfo(COPT.Param.TimeLimit)
```

Model.setBasis()

Synopsis

`setBasis(varbasis, constrbasis)`

Description

Set basis status for all variables and linear constraints in LP. The parameters `varbasis` and `constrbasis` are list objects whose number of elements is the total number of variables or linear constraints.

Arguments

`varbasis`

The basis status of variables.

`constrbasis`

The basis status of constraints.

Example

```
# Set basis status for all variables and linear constraints in the model.
m.setBasis(varbasis, constrbasis)
```

Model.setSlackBasis()

Synopsis

`setSlackBasis()`

Description

Set LP basis to be slack.

Example

```
#  Set LP basis to be slack.
m.setSlackBasis()
```

Model.setVarType()

Synopsis

`setVarType(vars, vartypes)`

Description

Set the type of specific variable.

If parameter `vars` is *Var Class* object, then parameter `vartypes` is Variable types constant;

If parameter `vars` is dictionary or *tupledict Class* object, then parameter `vartypes` can be Variable types constart, dictionary or *tupledict Class* object;

If parameter `vars` is list or *VarArray Class* object, then parameter `vartypes` can be Variable types constart or list object.

Arguments

`vars`

The specified variable.

`vartypes`

Type of the specified variable.

Example

```
# Set variable x as integer variable.
m.setVarType(x, COPT.INTEGER)
# Set variables x and y as binary variables.
m.setVarType([x, y], COPT.BINARY)
# Set the variables in tupledict object xdict as continuous variables.
m.setVarType(xdict, COPT.CONTINUOUS)
```

Model.setMipStart()

Synopsis

```
setMipStart(vars, startvals)
```

Description

Set initial value for specified variables, valid only for integer programming.

If parameter `vars` is *Var Class* object, then parameter `startvals` is constant; If parameter `vars` is dictionary or *tupledict Class* object, then parameter `startvals` can be constant, dictionary or *tupledict Class* object; If parameter `vars` is list or *VarArray Class* object, then parameter `startvals` can be constant or list object.

Notice: You may want to call this method several times to input the MIP start. Please call `loadMipStart()` once when the input is done.

Arguments

`vars`

The specified variable.

`startvals`

Initial value of the specified variable

Example

```
# Set initial value of x as 1.
m.setMipStart(x, 1)
# Set initial value of x, y as 2, 3.
m.setMipStart([x, y], [2, 3])
# Set initial value of all variables in tupledict xdict as 1.
m.setMipStart(xdict, 1)

# Load initial solution to model
m.loadMipStart()
```

Model.loadMipStart()

Synopsis

`loadMipStart()`

Description

Load the currently specified initial values into model.

Notice: After calling this method, the previously initial values will be cleared, and users can continue to set initial values for specified variables.

Model.setInfo()

Synopsis

`setInfo(infoname, args, newvals)`

Description

Set new information value for specific variables or constraints.

If parameter `args` is *Var Class* object or *Constraint Class* object, then parameter `newvals` is constant; If parameter `args` is dictionary or *tupledict Class* object, then parameter `newvals` can be constant, dictionary or *tupledict Class* object; If parameter `args` is list, *VarArray Class* object or *ConstrArray Class* object, then parameter `newvals` can be constant or list; If parameter `args` is *MVar Class* object or *MConstr Class* object, then parameter `newvals` can be constant or `numpy.ndarray`.

Arguments

`infoname`

The specified information name. The full list of available names can be found in *Information* section.

`args`

The specified variables of constraints.

`newvals`

Value of the specified information.

Example

```
m.setInfo(COPT.Info.LB, [x, y], [1.0, 2.0])

# Set the upperbound of variable x as 1.0
m.setInfo(COPT.Info.UB, x, 1.0)
# Set the lowerbound of variables x and y as 1.0, 2.0, respectively.
m.setInfo(COPT.Info.LB, [x, y], [1.0, 2.0])
# Set the objective of all variables in tupledict xdict as 0.
m.setInfo(COPT.Info.OBJ, xdict, 0.0)
```

Model.setParam()

Synopsis

```
setParam(paramname, newval)
```

Description

Set the value of a parameter to a specific value.

Arguments

`paramname`

The name of parameter to be set. The list of available names can be found in *Parameters* section.

`newval`

New value of parameter.

Example

```
# Set time limit of solving to 1 hour.
m.setParam(COPT.Param.TimeLimit, 3600)
```

Model.resetParam()

Synopsis

```
resetParam()
```

Description

Reset all parameters to their default values.

Example

```
# Reset all parameters to their default values.
m.resetParam()
```

Model.read()

Synopsis

```
read(filename)
```

Description

Determine the type of data by the file suffix and read it into a model.

Currently, it supports MPS files (suffix `'.mps'` or `'.mps.gz'`), LP files (suffix `'.lp'` or `'.lp.gz'`), SDPA files (suffix `'.dat-s'` or `'.dat-s.gz'`), CBF files (suffix `'.cbf'` or `'.cbf.gz'`), COPT binary format files (suffix `'.bin'`), basis files (suffix `'.bas'`), result files (suffix `'.sol'`), start files (suffix `'.mst'`), and parameter files (suffix `'.par'`).

Arguments

`filename`

Name of the file to be read.

Example

```
# Read MPS format model file
m.read('test.mps.gz')
# Read LP format model file
m.read('test.lp.gz')
# Read COPT binary format model file
m.read('test.bin')
# Read basis file
m.read('testlp.bas')
# Read solution file
m.read('testmip.sol')
# Read start file
m.read('testmip.mst')
# Read paramter file
m.read('test.par')
```

Model.readMps()

Synopsis

`readMps(filename)`

Description

Read MPS file to model.

Arguments

`filename`

The name of the MPS file to be read.

Example

```
# Read file "test.mps.gz" according to mps file format
m.readMps('test.mps.gz')
# Read file "test.lp.gz" according mps file format
m.readMps('test.lp.gz')
```

Model.readLp()

Synopsis

`readLp(filename)`

Description

Read a file to model according to LP file format.

Arguments

`filename`

Name of the LP file to be read.

Example

```
# Read file"test.mps.gz" according to LP file format
m.readLp('test.mps.gz')
# Read file"test.lp.gz" according to LP file format
m.readLp('test.lp.gz')
```

Model.readSdpa()

Synopsis

```
readSdpa(filename)
```

Description

Read a file to model according to SDPA file format.

Arguments

`filename`

Name of the SDPA file to be read.

Example

```
# Read file"test.dat-s" according to SDPA file format
m.readSdpa('test.dat-s')
```

Model.readCbf()

Synopsis

```
readCbf(filename)
```

Description

Read a file to model according to CBF file format.

Arguments

`filename`

Name of the CBF file to be read.

Example

```
# Read file"test.cbf" according to CBF file format
m.readCbf('test.cbf')
```

Model.readBin()

Synopsis

```
readBin(filename)
```

Description

Read COPT binary format file to model.

Arguments

`filename`

The name of the COPT binary format file to be read.

Example

```
# Read file "test.bin" according COPT binary file format
m.readBin('test.bin')
```

Model.readSol()

Synopsis

```
readSol(filename)
```

Description

Read a file to model according to solution file format.

Notice: The default solution value is 0, i.e. a partial solution will be automatically filled in with zeros. If read successfully, then the values read can be act as initial solution for integer programming.

Arguments

`filename`

Name of file to be read.

Example

```
# Read file "testmip.sol" according to solution file format.
m.readSol('testmip.sol')
# Read file testmip.txt" according to solution file format.
m.readSol('testmip.txt')
```

Model.readBasis()

Synopsis

```
readBasis(filename)
```

Description

Read basis status of variables and linear constraints to model accoring to basis solution format, valid only for linear programming.

Arguments

`filename`

The name of the basis file to be read.

Example

```
# Read file "testmip.bas" to basis solution format
m.readBasis('testmip.bas')
# Read file "testmip.txt" to basis solution format
m.readBasis('testmip.txt')
```

Model.readMst()

Synopsis

```
readMst(filename)
```

Description

Read initial solution data to model according to initial solution file format.

Notice: If read successfully, the read value will be act as initial solution for integer programming model. Variable values may not be specified completely, if the value of variable is specified for multiple times, the last specified value is used.

Arguments

`filename`

Name of the file to be read.

Example

```
# Read file "testmip.mst" according to initial solution file format.
m.readMst('testmip.mst')
# Read file "testmip.txt" according to initial solution file format.
m.readMst('testmip.txt')
```

Model.readParam()

Synopsis

`readParam(filename)`

Description

Read optimization parameters to model according to parameter file format.

Notice: If any optimization parameter is specified multiple times, the last spcified value is used.

Arguments

`filename`

The name of the parameter file to be read.

Example

```
# Read file "testmip.par" according to parameter file format.
m.readParam('testmip.par')
# Read file "testmip.txt" according to parameter file format.
m.readParam('testmip.txt')
```

Model.readTune()

Synopsis

`readTune(filename)`

Description

Read the tuning parameters and combine them into the model according to the tuning file format.

Arguments

`filename`

The name of the file to be read.

Example

Model.write()

Synopsis

`write(filename)`

Description

Currently, COPT supports writing of MPS files (suffix `'.mps'`), LP files (suffix `'.lp'`), CBF files (suffix `'.cbf'`), COPT binary format files (suffix `'.bin'`), basis files (suffix `'.bas'`), LP solution files (suffix `'.sol'`), initial solution files (suffix `'.mst'`), and parameter files (suffix `'.par'`).

Arguments

`filename`

The file name to be written.

Example

```
# Write MPS file
m.write('test.mps')
# Write LP file
m.write('test.lp')
# Write COPT binary format file
m.write('test.bin')
# Write basis file
m.write('testlp.bas')
# Write solution file
m.write('testmip.sol')
# Write initial solution file
m.write('testmip.mst')
# Write parameter file
m.write('test.par')
```

Model.writeMps()

Synopsis

`writeMps(filename)`

Description

Write current model into an MPS file.

Arguments

`filename`

The name of the MPS file to be written.

Example

```
# Write MPS model file "test.mps"
m.writeMps('test.mps')
```

Model.writeMpsStr()

Synopsis

```
writeMpsStr()
```

Description

Write current model into a buffer as MPS format.

Example

```
# Write model to buffer 'buff' and print model
buff = m.writeMpsStr()
print(buff.getData())
```

Model.writeLp()

Synopsis

```
writeLp(filename)
```

Description

Write current optimization model to a LP file.

Arguments

`filename`

The name of the LP file to be written.

Example

```
# Write LP model file "test.lp"
m.writeLp('test.lp')
```

Model.writeCbf()

Synopsis

```
writeCbf(filename)
```

Description

Write current optimization model to a CBF file.

Arguments

`filename`

The name of the CBF file to be written.

Example

```
# Write CBF model file "test.cbf"
m.writeCbf('test.cbf')
```

Model.writeBin()

Synopsis

```
writeBin(filename)
```

Description

Write current model into an COPT binary format file.

Arguments

`filename`

The name of the COPT binary format file to be written.

Example

```
# Write COPT binary format model file "test.bin"
m.writeBin('test.bin')
```

Model.writeIIS()

Synopsis

```
writeIIS(filename)
```

Description

Write current irreducible inconsistent subsystem into an IIS file.

Arguments

`filename`

The name of the IIS file to be written.

Example

```
# Write IIS file "test.iis"
m.writeIIS('test.iis')
```

Model.writeRelax()

Synopsis

```
writeRelax(filename)
```

Description

Write the feasibility relaxation model into a Relax file.

Arguments

`filename`

The name of the Relax file to be written.

Example

```
# Write Relax file "test.relax"
m.writeRelax('test.relax')
```

Model.writeSol()

Synopsis

```
writeSol(filename)
```

Description

Output the model solution to a solution file.

Arguments

`filename`

The name of the solution file to be written.

Example

```
# Write solution file "test.sol"
m.writeSol('test.sol')
```

Model.writePoolSol()

Synopsis

```
writePoolSol(isol, filename)
```

Description

Output selected pool solution to a solution file.

Arguments

`isol`

Index of pool solution.

`filename`

The name of the solution file to be written.

Example

```
# Write 1-th pool solution to solution file "poolsol_1.sol"
m.writePoolSol('poolsol_1.sol')
```

Model.writeBasis()

Synopsis

```
writeBasis(filename)
```

Description

Write the LP basic solution to a basis file.

Arguments

`filename`

The name of the basis file to be written.

Example

```
# Write the basis file "testlp.bas"
m.writeBasis('testlp.bas')
```

Model.writeMst()

Synopsis

```
writeMst(filename)
```

Description For integer programming models, write the best integer solution currently to the initial solution file. If there are no integer solutions, then the first set of initial solution stored in model is output.

Arguments

`filename`

Name of the file to be written.

Example

```
# Output initial solution file "testmip.mst"
m.writeMst('testmip.mst')
```

Model.writeParam()

Synopsis

```
writeParam(filename)
```

Description

Output modified parameters to a parameter file.

Arguments

`filename`

The name of the parameter file to be written.

Example

```
# Output parameter file "testmip.par"
m.writeParam('testmip.par')
```

Model.writeTuneParam()

Synopsis

```
writeTuneParam(idx, filename)
```

Description

Output the parameter tuning result of the specified number to the parameter file.

Arguments

`idx`

Parameter tuning result number.

`filename`

The name of the parameter file to be output.

Example

Model.setLogFile()

Synopsis

`setLogFile(logfile)`

Description

Set the optimizer log file.

Arguments

`logfile`

The log file.

Example

```
# Set the log file as "copt.log"
m.setLogFile('copt.log')
```

Model.setLogCallback()

Synopsis

`setLogCallback(logcb)`

Description

Set the call back function of log.

Arguments

`logcb`

Call back function of log.

Example

```
# Set the call back function of log as a python function 'logcbfun'.
m.setLogCallback(logcbfun)
```

Model.solve()

Synopsis

`solve()`

Description

Solve an optimization problem.

Example

```
# Solve the model.
m.solve()
```

Model.solveLP()

Synopsis

```
solveLP()
```

Description

Solve LP model. If the model is integer programming, then the model will be solved as LP.

Example

```
# Solve a model calling LP solver.
m.solveLP()
```

Model.computeIIS()

Synopsis

```
computeIIS()
```

Description

Compute IIS for infeasible model.

Example

```
# Compute IIS for infeasible model.
m.computeIIS()
```

Model.feasRelax()

Synopsis

```
feasRelax(vars, lbpen, ubpen, constrs, rhspen, uppen=None)
```

Description

Compute the feasibilty relaxation of an infeasible model

Arguments

`vars`

Variables to relax.

`lbpen`

The penalty relating to lower bounds. If `None`, no lower bound violations are allowed. If a variable's penalty is `COPT.INFINITY`, lower bound violation is not allowed on it.

`ubpen`

The penalty relating to upper bounds. If `None`, no upper bound violations are allowed. If a variable's penalty is `COPT.INFINITY`, upper bound violation is not allowed on it.

`constrs`

Constraints to relax.

`rhspen`

The penalty relating to constraints. If `None`, no constraint violations are allowed. If a constraint's penalty is `COPT.INFINITY`, it's not allowed to be violated.

`uppen`

The penalty relating to the upper bound of bilateral constraints. If `None`, specified by `rhspen`; If a constraint's penalty is `COPT.INFINITY`, constraint upper bound violation is not allowed on it.

Example

```
# compute feasibility relaxation for model m
m.feasRelax(vars, lbpen, ubpen, constrs, rhspen)
```

Model.feasRelaxS()

Synopsis

`feasRelaxS(vrelax, crelax)`

Description

Compute the feasibilty relaxation of an infeasible model

Arguments

`vrelax`

Whether to relax variables.

`crelax`

Whether to relax constraints.

Example

```
# Compute the feasibilty relaxation of model m
m.feasRelaxS(True, True)
```

Model.tune()

Synopsis

`tune()`

Description

Parameter tuning of the model.

Example

Model.loadTuneParam()

Synopsis

`loadTuneParam(idx)`

Description

Load the parameter tuning results of the specified number into the model.

Example

Model.interrupt()

Synopsis

```
interrupt()
```

Description

Interrupt solving process of current problem.

Example

```
# Interrupt the solving process.
m.interrupt()
```

Model.remove()

Synopsis

```
remove(args)
```

Description

Remove variables or constraints from a model.

To remove variable, then parameter `args` can be *Var Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

To remove linear constraint, then parameter `args` can be *Constraint Class* object, *ConstrArray Class* object, list, dictionary or *tupledict Class* object.

To remove SOS constraint, then parameter `args` can be *SOS Class* object, *SOSArray Class* object, list, dictionary or *tupledict Class* object.

To remove Second-Order-Cone constraints, then parameter `args` can be *Cone Class* object, *ConeArray Class* object, list, dictionary or *tupledict Class* object.

To remove quadratic constraints, then parameter `args` can be *QConstraint Class* object, *QConstrArray Class* object, list, dictionary or *tupledict Class* object.

To remove positive semi-definite constraints, then parameter `args` can be *PsdConstraint Class* object, *PsdConstrArray Class* object, list, dictionary or *tupledict Class* object.

To remove Indicator constraint, then parameter `args` can be *GenConstr Class* object, *GenConstrArray Class* object, list, dictionary or *tupledict Class* object.

Arguments

`args`

Variables or constraints to be removed.

Example

```
# Remove linear constraint conx
m.remove(conx)
# Remove variables x and y
m.remove([x, y])
```

Model.reset()

Synopsis

```
reset(clearall=0)
```

Description Reset the model to an unsolved state, which means resetting previously solution information. If parameter `clearall` is 1, the initial solution information is also resetted.

Arguments

`clearall`

Whether to reset initial solution information. Optional, 0 by default, which means not resetting initial solution information.

Example

```
# Reset the solution information in model.
m.reset()
```

Model.clear()

Synopsis

```
clear()
```

Description

Clear the model.

Example

```
# Clear the contents in model.
m.clear()
```

Model.clone()

Synopsis

```
clone()
```

Description

Create a deep copy of an existing model. Return a *Model Class* object.

Example

```
# Create a deep copy of model
mcopy = m.clone()
```

Model.setCallback()

Synopsis

```
setCallback(cb, cbctx)
```

Description

Set the callback function of the model.

Arguments

`cb`

Callback Class object.

`cbctx`

Callback context. Please refer to *Callback context* .

Example

```
cb = CoptCallback()
model.setCallback(cb, COPT.CBCONTEXT_MIPSOL)
```

21.2.4 Var Class

For easy access to information of variables, Var object provides methods such as `Var.LB`. The full list of information can be found in the *Information* section. For convenience, information can be accessed by names in original case or lowercase.

In addition, you can also access the value of the variable through `Var.x`, the variable type through `Var.vtype`, the name of the variable through `Var.name`, the Reduced cost value of the variable in LP through `Var.rc`, the basis status through `Var.basis`, and the index of the variable in the coefficient matrix through `Var.index` the Reduced cost value of the variable in LP through `Var.rc`, the basis status through `Var.basis`, and the index of the variable in the coefficient matrix through `Var.index`.

For the model-related information of the variables, as well as the variable type and name, the user can set the corresponding information value in the form of `"Var.LB = 0.0"`.

Var object contains related operations of COPT variables and provides the following methods:

Var.getType()

Synopsis

`getType()`

Description

Retrieve the type of variable.

Example

```
# Retrieve the type of variable v
vtype = v.getType()
```

Var.getName()

Synopsis

`getName()`

Description

Retrieve the name of variable.

Example

```
# Retrieve the name of variable v
varname = v.getName()
```

Var.getBasis()

Synopsis

getBasis()

Description

Retrieve the basis status of variable.

Example

```
# Retrieve the basis status of variable v
varbasis = v.getBasis()
```

Var.getLowerIIS()

Synopsis

getLowerIIS()

Description

Retrieve the IIS status of lower bound of variable.

Example

```
# Retrieve the IIS status of lower bound of variable v
lowerIIS = v.getLowerIIS()
```

Var.getUpperIIS()

Synopsis

getUpperIIS()

Description

Retrieve the IIS status of upper bound of variable.

Example

```
# Retrieve the IIS status of upper bound of variable v
upperIIS = v.getUpperIIS()
```

Var.getIdx()

Synopsis

getIdx()

Description

Retrieve the subscript of the variable in the coefficient matrix.

Example

```
#  Retrieve the subscript of variable v
vindex = v.getIdx()
```

Var.setType()

Synopsis

`setType(newtype)`

Description

Set the type of variable.

Arguments

`newtype`

The type of variable to be set. Please refer to Variable types section for possible values.

Example

```
# Set the type of variable v
v.setType(COPT.BINARY)
```

Var.setName()

Synopsis

`setName(newname)`

Description

Set the name of variable.

Arguments

`newname`

The name of variable to be set.

Example

```
# Set the name of variable v
v.setName(COPT.BINARY)
```

Var.getInfo()

Synopsis

`getInfo(infoname)`

Description

Retrieve specified information. Return a constant.

Arguments

`infoname`

The name of the information. Please refer to *Information* for possible values.

Example

```
# Get lowerbound of variable x
x.getInfo(COPT.Info.LB)
```

Var.setInfo()

Synopsis

```
setInfo(infoname, newval)
```

Description

Set new information value for a variable.

Arguments

`infoname`

The name of the information to be set. Please refer to *Information* section for possible values.

`newval`

New information value to set.

Example

```
# Set the lower bound of variable x
x.setInfo(COPT.Info.LB, 1.0)
```

Var.remove()

Synopsis

```
remove()
```

Description

Delete the variable from model.

Example

```
# Delete variable 'x'
x.remove()
```

21.2.5 VarArray Class

To facilitate users to operate on multiple *Var Class* objects, the Python interface of COPT provides VarArray object with the following methods:

VarArray()

Synopsis

```
VarArray(vars=None)
```

Description

Create a *VarArray Class* object.

If parameter `vars` is `None`, then create an empty *VarArray Class* object, otherwise initialize the new created *VarArray Class* object based on `vars`.

Arguments

`vars`

Variables to be added. Optional, `None` by default. `vars` can be *Var Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty VarArray object
vararr = VarArray()
# Create an empty VarArray object and initialize variables x, y.
vararr = VarArray([x, y])
```

VarArray.pushBack()

Synopsis

pushBack(vars)

Description

Add single or multiple *Var Class* objects.

Arguments

vars

Variables to be applied. `vars` can be *Var Class* object, *VarArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add variable x to vararr
vararr.pushBack(x)
# Add variables x and y to vararr
vararr.pushBack([x, y])
```

VarArray.getVar()

Synopsis

getVar(idx)

Description

Retrieve a variable from an index in a *VarArray Class* object. Return a *Var Class* object.

Arguments

idx

Subscript of the specified variable in *VarArray Class* object, starting with 0.

Example

```
# Get the variable with subscript of 1 in vararr
 vararr.getVar(1)
```

VarArray.getAll()

Synopsis

```
getAll()
```

Description

Retrieve all variables in *VarArray Class* object. Returns a list object.

Example

```
# Get all variables in 'vararr'
varall = vararr.getAll()
```

VarArray.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of variables in *VarArray Class* object.

Example

```
# Retrieve the number of variables in vararr.
arrsize = vararr.getSize()
```

21.2.6 PsdVar Class

PsdVar object contains related operations of COPT positive semi-definite variables and provides the following methods:

PsdVar.getName()

Synopsis

```
getName()
```

Description

Retrieve the name of positive semi-definite variable.

Example

```
# Retrieve the name of variable v
varname = v.getName()
```

PsdVar.getIdx()

Synopsis

```
getIdx()
```

Description

Retrieve the subscript of the variable in the model.

Example

```
# Retrieve the subscript of variable v
vindex = v.getIdx()
```

PsdVar.getDim()

Synopsis

```
getDim()
```

Description

Retrieve the dimension of positive semi-definite variable.

Example

```
# Retrieve the dimension of variable "v"
vdim = v.getDim()
```

PsdVar.getLen()

Synopsis

```
getLen()
```

Description

Retrieve the length of the expanded positive semi-definite variable.

Example

```
# Retrieve the length of the expanded positive semi-definite variable "v"
vlen = v.getLen()
```

PsdVar.setName()

Synopsis

```
setName(newname)
```

Description

Set the name of positive semi-definite variable.

Arguments

`newname`

The name of positive semi-definite variable to be set.

Example

```
# Set the name of variable v
v.setName('v')
```

PsdVar.getInfo()

Synopsis

```
getInfo(infoname)
```

Description

Retrieve specified information of positive semi-definite variable. Return a list.

Arguments

`infoname`

The name of the information. Please refer to *Information* for possible values.

Example

```
# Get solution values of positive semi-definite variable x
sol = x.getInfo(COPT.Info.Value)
```

PsdVar.remove()

Synopsis

```
remove()
```

Description

Delete the positive semi-definite variable from model.

Example

```
# Delete variable 'x
x.remove()
```

21.2.7 PsdVarArray Class

To facilitate users to operate on multiple *PsdVar Class* objects, the Python interface of COPT provides PsdVarArray object with the following methods:

PsdVarArray()

Synopsis

```
PsdVarArray(vars=None)
```

Description

Create a *PsdVarArray Class* object.

If parameter `vars` is `None`, then create an empty *PsdVarArray Class* object, otherwise initialize the new created *PsdVarArray Class* object based on `vars`.

Arguments

`vars`

Positive semi-definite variables to be added. Optional, `None` by default. `vars` can be *PsdVar Class* object, *PsdVarArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty PsdVarArray object
vararr = PsdVarArray()
# Create a PsdVarArray object containing positive semi-definite variables x, y.
vararr = PsdVarArray([x, y])
```

PsdVarArray.pushBack()

Synopsis

```
pushBack(var)
```

Description

Add single or multiple *PsdVar Class* objects.

Arguments

`var`

Postive semi-definite variables to be applied. `vars` can be *PsdVar Class* object, *PsdVarArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add variable x to vararr
vararr.pushBack(x)
# Add variables x and y to vararr
vararr.pushBack([x, y])
```

PsdVarArray.getPsdVar()

Synopsis

```
getPsdVar(idx)
```

Description

Retrieve a positive semi-definite variable from an index in a *PsdVarArray Class* object. Return a *PsdVar Class* object.

Arguments

`idx`

Subscript of the specified positive semi-definite variable in *PsdVarArray Class* object, starting with 0.

Example

```
# Get the positive semi-definite variable with subscript of 1 in vararr
var = vararr.getPsdVar(1)
```

PsdVarArray.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of positive semi-definite variables in *PsdVarArray Class* object.

Example

```
# Retrieve the number of variables in vararr.
arrsize = vararr.getSize()
```

21.2.8 SymMatrix Class

SymMatrix object contains related operations of COPT symmetric matrices and provides the following methods:

SymMatrix.getIdx()

Synopsis

```
getIdx()
```

Description

Retrieve the subscript of the symmetric matrix in the model.

Example

```
# Retrieve the subscript of symmetric matrix mat
matidx = mat.getIdx()
```

SymMatrix.getDim()

Synopsis

```
getDim()
```

Description

Retrieve the dimension of symmetric matrix.

Example

```
# Retrieve the dimension of symmetric matrix "mat".
matdim = mat.getDim()
```

21.2.9 SymMatrixArray Class

To facilitate users to operate on multiple *SymMatrix Class* objects, the Python interface of COPT provides SymMatrixArray object with the following methods:

SymMatrixArray()

Synopsis

`SymMatrixArray(mats=None)`

Description

Create a *SymMatrixArray Class* object.

If parameter `mats` is `None`, then create an empty *SymMatrixArray Class* object, otherwise initialize the new created *SymMatrixArray Class* object based on `mats`.

Arguments

`mats`

`mats` can be *SymMatrix Class* object, *SymMatrixArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty SymMatrixArray object
matarr = SymMatrixArray()
# Create a SymMatrixArray object containing matx, maty.
matarr = SymMatrixArray([matx, maty])
```

SymMatrixArray.pushBack()

Synopsis

`pushBack(mat)`

Description

Add single or multiple *SymMatrix Class* objects.

Arguments

`mat`

Symmetric matrices to be applied. `mat` can be *SymMatrix Class* object, *SymMatrixArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add symmetric matrix matx to matarr
matarr.pushBack(matx)
# Add symmetric matrices matx and maty to matarr
matarr.pushBack([matx, maty])
```

SymMatrixArray.getMatrix()

Synopsis

```
getMatrix(idx)
```

Description

Retrieve a symmetric matrix from an index in a *SymMatrixArray Class* object. Return a *SymMatrix Class* object.

Arguments

`idx`

Subscript of the specified symmetric matrix in *SymMatrixArray Class* object, starting with 0.

Example

```
# Get the symmetric matrix with subscript of 1 in matarr
mat = matarr.getMatrix(1)
```

SymMatrixArray.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of symmetric matrices in *SymMatrixArray Class* object.

Example

```
# Retrieve the number of symmetric matrices in matarr.
arrsize = matarr.getSize()
```

21.2.10 Constraint Class

For easy access to information of constraints, Constraint class provides methods such as `Constraint.LB`. The supported information can be found in *Information* section. For convenience, information can be queried by names in original case or lowercase.

In addition, you can access the name of the constraint through `Constraint.name`, the dual value of the constraint in LP through `Constraint.pi`, the basis status of the constraint through `Constraint.basis`, and the index in the coefficient matrix through `Constraint.index`.

For the model-related information and constraint name, the user can also set the corresponding information in the form of `"Constraint.lb = -100"`.

Constraint object contains related operations of COPT constraints and provides the following methods:

Constraint.getName()

Synopsis

```
getName()
```

Description

Retrieve the name of linear constraint.

Example

```
# Retrieve the name of linear constraint 'con'.
conname = con.getName()
```

Constraint.getBasis()

Synopsis

```
getBasis()
```

Description

Retrieve the basis status of linear constraint.

Example

```
# Retrieve the basis status of linear constraint 'con'.
conbasis = con.getBasis()
```

Constraint.getLowerIIS()

Synopsis

```
getLowerIIS()
```

Description

Retrieve the IIS status of lower bound of linear constraint.

Example

```
# Retrieve the IIS status of lower bound of linear constraint 'con'.
lowerIIS = con.getLowerIIS()
```

Constraint.getUpperIIS()

Synopsis

```
getUpperIIS()
```

Description

Retrieve the IIS status of upper bound of linear constraint.

Example

```
# Retrieve the IIS status of upper bound of linear constraint 'con'.
upperIIS = con.getUpperIIS()
```

Constraint.getIdx()

Synopsis

`getIdx()`

Description

Retrieve the subscript of linear constraint in coefficient matrix.

Example

```
# Retrieve the subscript of linear constraint con.
conidx = con.getIdx()
```

Constraint.setName()

Synopsis

`setName(newname)`

Description

Set the name of linear constraint.

Arguments

`newname`

The name of constraint to be set.

Example

```
# Set the name of linear constraint 'con'.
con.setName('con')
```

Constraint.getInfo()

Synopsis

`getInfo(infoname)`

Description

Retrieve specified information. Return a constant.

Arguments

`infoname`

Name of the information to be obtained. Please refer to *Information* section for possible values.

Example

```
# Get the lower bound of linear constraint con
conlb = con.getInfo(COPT.Info.LB)
```

Constraint.setInfo()

Synopsis

```
setInfo(infoname, newval)
```

Description

Set new information value to the specified constraint.

Arguments

`infoname`

The name of the information to be set. Please refer to *Information* section for possible values.

`newval`

New information value to be set.

Example

```
# Set the lower bound of linear constraint con
con.setInfo(COPT.Info.LB, 1.0)
```

Constraint.remove()

Synopsis

```
remove()
```

Description

Delete the linear constraint from model.

Example

```
# Delete the linear constraint 'conx'
conx.remove()
```

21.2.11 ConstrArray Class

To facilitate users to operate on multiple *Constraint Class* objects, the Python interface of COPT provides ConstrArray class with the following methods:

ConstrArray()

Synopsis

```
ConstrArray(constrs=None)
```

Description

Create a *ConstrArray Class* object.

If parameter `constrs` is `None`, the create an empty *ConstrArray Class* object, otherwise initialize the newly created *ConstrArray Class* object with parameter `constrs`

Arguments

`constrs`

Linear constraints to be added. `None` by default.

`constrs` can be *Constraint Class* object, *ConstrArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty ConstrArray object
conarr = ConstrArray()
# Create an ConstrArray object initialized with linear constraint conx and cony
conarr = ConstrArray([conx, cony])
```

ConstrArray.pushBack()

Synopsis

`pushBack(constrs)`

Description

Add single or multiple *Constraint Class* objects.

Arguments

`constrs`

Linear constraints to be applied. `constrs` can be *Constraint Class* object, *ConstrArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add linear constraint r to conarr
conarr.pushBack(r)
# Add linear constraint r0 and r1 to conarr
conarr.pushBack([r0, r1])
```

ConstrArray.getConstr()

Synopsis

`getConstr(idx)`

Description

Retrieve the linear constraint according to its subscript in *ConstrArray Class* object. Return a *Constraint Class* object.

Arguments

`idx`

Subscript of the desired constraint in *ConstrArray Class* object, starting with 0.

Example

```
# Retrieve the linear constraint with subscript 1  in conarr
conarr.getConstr(1)
```

ConstrArray.getAll()

Synopsis

```
getAll()
```

Description

Retrieve all linear constraints in *ConstrArray Class* object. Returns a list object.

Example

```
# Get all linear constraints in 'conarr'
cons = conarr.getAll()
```

ConstrArray.getSize()

Synopsis

```
getSize()
```

Description

Get the number of elements in *ConstrArray Class* object.

Example

```
# Get the number of linear constraints in conarr
arrsize = conarr.getSize()
```

21.2.12 ConstrBuilder Class

ConstrBuilder object contains operations related to temporary constraints when building constraints, and provides the following methods:

ConstrBuilder()

Synopsis

```
ConstrBuilder()
```

Description

Create an empty *ConstrBuilder Class* object

Example

```
# Create an empty linear constraint builder
constrbuilder = ConstrBuilder()
```

ConstrBuilder.setBuilder()

Synopsis

```
setBuilder(expr, sense)
```

Description

Set expression and constraint type for linear constraint builder.

Arguments

```
expr
```

The expression to be set, which can be *Var Class* expression or *LinExpr Class* expression.

`sense`

Sense of constraint. The full list of available tyeps can be found in *Constraint type* section.

Example

```
# Set the expresson of linear constraint builder as: x+y-1, and sense of constraint as equal
constrbuilder.setBuilder(x + y - 1, COPT.EQUAL)
```

ConstrBuilder.getExpr()

Synopsis

`getExpr()`

Description

Retrieve the expression of a linear constraint builder object.

Example

```
#  Retrieve the expression of a linear constraint builder
linexpr = constrbuilder.getExpr()
```

ConstrBuilder.getSense()

Synopsis

`getSense()`

Description

Retrieve the constraint sense of linear constraint builder object.

Example

```
# Retrieve the constraint sense of linear constraint builder object.
consense = constrbuilder.getSense()
```

21.2.13 ConstrBuilderArray Class

To facilitate users to operate on multiple *ConstrBuilder Class* objects, the Python interface of COPT provides ConstrArray object with the following methods:

ConstrBuilderArray()

Synopsis

`ConstrBuilderArray(constrbuilders=None)`

Description

Create a *ConstrBuilderArray Class* object.

If parameter `constrbuilders` is `None`, then create an empty *ConstrBuilderArray Class* object, otherwise initialize the newly created *ConstrBuilderArray Class* object by parameter `constrbuilders`.

Arguments

`constrbuilders` Linear constraint builder to be added. Optional, `None` by default. It can be *ConstrBuilder Class* object, *ConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty ConstrBuilderArray object.
conbuilderarr = ConstrBuilderArray()
# Create a ConstrBuilderArray object and initialize it with builders: conbuilderx and␣
↪conbuildery
conbuilderarr = ConstrBuilderArray([conbuilderx, conbuildery])
```

ConstrBuilderArray.pushBack()

Synopsis

`pushBack(constrbuilder)`

Description

Add single or multiple *ConstrBuilder Class* objects.

Arguments

`constrbuilder`

Builder of linear constraint to be added, which can be *ConstrBuilder Class* object, *ConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add linear constraint builder conbuilderx to conbuilderarr
conbuilderarr.pushBack(conbuilderx)
# Add linear constraint builders conbuilderx and conbuildery to conbuilderarr
conbuilderarr.pushBack([conbuilderx, conbuildery])
```

ConstrBuilderArray.getBuilder()

Synopsis

`getBuilder(idx)`

Description

Retrieve a temporary constraint from its index in *ConstrBuilderArray Class* object. Return a *ConstrBuilder Class* object.

Retrieve the corresponding builder object according to the subscript of linear constraint builder in *ConstrBuilderArray Class* object.

Arguments

`idx`

Subscript of the linear constraint builder in the *ConstrBuilderArray Class* object, starting with 0.

Example

```
# Retrieve the builder with subscript 1 in conbuilderarr
conbuilder = conbuilderarr.getBuilder(1)
```

ConstrBuilderArray.getSize()

Synopsis

```
getSize()
```

Description

Get the number of elements in *ConstrBuilderArray Class* object.

Example

```
# Get the number of builders in conbuilderarr
arrsize = conbuilderarr.getSize()
```

21.2.14 QConstraint Class

For easy access to information of quadratic constraints, QConstraint class provides methods such as `QConstraint.index`. The supported information can be found in *Information* section. For convenience, information can be queried by names in original case or lowercase.

In addition, you can access the name of the quadratic constraint through `QConstraint.name`, and the index in the model through `QConstraint.index`.

For the model-related information and constraint name, the user can also set the corresponding information in the form of `"QConstraint.rhs = -100"`.

QConstraint object contains related operations of COPT quadratic constraints and provides the following methods:

QConstraint.getName()

Synopsis

```
getName()
```

Description

Retrieve the name of quadratic constraint.

Example

```
# Retrieve the name of quadratic constraint 'qcon'
qconname = qcon.getName()
```

QConstraint.getRhs()

Synopsis

```
getRhs()
```

Description

Retrieve the right hand side of quadratic constraint.

Example

```
# Retrieve the RHS of quadratic constraint 'qcon'
qconrhs = qcon.getRhs()
```

QConstraint.getSense()

Synopsis

```
getSense()
```

Description

Retrieve the type of quadratic constraint.

Example

```
# Retrieve the type of quadratic constraint 'qcon'
qconsense = qcon.getSense()
```

QConstraint.getIdx()

Synopsis

```
getIdx()
```

Description

Retrieve the subscript of quadratic constraint.

Example

```
# Retrieve the subscript of quadratic constraint 'qcon'
qconidx = qcon.getIdx()
```

QConstraint.setName()

Synopsis

```
setName(newname)
```

Description

Set the name of quadratic constraint.

Arguments

`newname`

The name of quadratic constraint to be set.

Example

```
# Set the name of quadratic constraint 'qcon'.
qcon.setName('qcon')
```

QConstraint.setRhs()

Synopsis

```
setRhs(rhs)
```

Description

Set the right hand side of quadratic constraint.

Arguments

`rhs`

The right hand side of quadratic constraint to be set.

Example

```
# Set the RHS of quadratic constraint 'qcon' to 0.0
qcon.setRhs(0.0)
```

QConstraint.setSense()

Synopsis

```
setSense(sense)
```

Description

Set the sense of quadratic constraint.

Arguments

`sense`

The sense of quadratic constraint to be set.

Example

```
# Set the sense of quadratic constraint 'qcon' to <=
qcon.setSense(COPT.LESS_EQUAL)
```

QConstraint.getInfo()

Synopsis

```
getInfo(infoname)
```

Description

Retrieve specified information. Return a constant.

Arguments

`infoname`

Name of the information to be obtained. Please refer to *Information* section for possible values.

Example

```
# Get the row activity of quadratic constraint 'qcon'
qconlb = qcon.getInfo(COPT.Info.Slack)
```

QConstraint.setInfo()

Synopsis

```
setInfo(infoname, newval)
```

Description

Set new information value to the specified quadratic constraint.

Arguments

`infoname`

The name of the information to be set. Please refer to *Information* section for possible values.

`newval`

New information value to be set.

Example

```
# Set the lower bound of quadratic constraint 'qcon'
qcon.setInfo(COPT.Info.LB, 1.0)
```

Constraint.remove()

Synopsis

`remove()`

Description

Delete the quadratic constraint from model.

Example

```
# Delete the quadratic constraint 'qconx'
qconx.remove()
```

21.2.15 QConstrArray Class

To facilitate users to operate on multiple *QConstraint Class* objects, the Python interface of COPT provides QConstrArray class with the following methods:

QConstrArray()

Synopsis

`QConstrArray(qconstrs=None)`

Description

Create a *QConstrArray Class* object.

If parameter `qconstrs` is `None`, the create an empty *QConstrArray Class* object, otherwise initialize the newly created *QConstrArray Class* object with parameter `qconstrs`

Arguments

`qconstrs`

Quadratic constraints to be added. `None` by default.

`qconstrs` can be *QConstraint Class* object, *QConstrArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty QConstrArray object
qconarr = QConstrArray()
# Create an QConstrArray object initialized with quadratic constraint qconx and qcony
qconarr = QConstrArray([qconx, qcony])
```

QConstrArray.pushBack()

Synopsis

`pushBack(constr)`

Description

Add single or multiple *QConstraint Class* object.

Arguments

`constr`

Quadratic constraints to be added. `None` by default.

`qconstrs` can be *QConstraint Class* object, *QConstrArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add quadratic constraint qr to conarr
qconarr.pushBack(qr)
# Add quadratic constraint qr0 and qr1 to qconarr
qconarr.pushBack([qr0, qr1])
```

QConstrArray.getQConstr()

Synopsis

`getQConstr(idx)`

Description

Retrieve the quadratic constraint according to its subscript in *QConstrArray Class* object. Return a *QConstraint Class* object.

Arguments

`idx`

Subscript of the desired quadratic constraint in *QConstrArray Class* object, starting with 0.

Example

```
# Retrieve the quadratic constraint with subscript 1  in qconarr
qcon = qconarr.getQConstr(1)
```

QConstrArray.getSize()

Synopsis

`getSize()`

Description

Get the number of elements in *QConstrArray Class* object.

Example

```
# Get the number of quadratic constraints in qconarr
qarrsize = qconarr.getSize()
```

21.2.16 QConstrBuilder Class

QConstrBuilder object contains operations related to temporary constraints when building quadratic constraints, and provides the following methods:

QConstrBuilder()

Synopsis

`QConstrBuilder()`

Description

Create an empty *QConstrBuilder Class* object.

Example

```
# Create an empty quadratic constraint builder
qconstrbuilder = QConstrBuilder()
```

QConstrBuilder.setBuilder()

Synopsis

`setBuilder(expr, sense, rhs)`

Description

Set expression, constraint type and RHS for quadratic constraint builder.

Arguments

`expr`

The expression to be set, which can be *Var Class* object, *LinExpr Class* object or *QuadExpr Class* object.

`sense`

Sense of quadratic constraint. The full list of available tyeps can be found in *Constraint type* section.

`rhs`

Right hand side of quadratic constraint.

Example

```
# Set the expresson of quadratic constraint builder as: x+y, sense of constraint as equal and␣
↪RHS as 1
qconstrbuilder.setBuilder(x + y, COPT.LESS_EQUAL, 1.0)
```

QConstrBuilder.getQuadExpr()

Synopsis

`getQuadExpr()`

Description

Retrieve the expression of a quadratic constraint builder object.

Example

```
#  Retrieve the expression of a quadratic constraint builder
quadexpr = constrbuilder.getQuadExpr()
```

QConstrBuilder.getSense()

Synopsis

```
getSense()
```

Description

Retrieve the constraint sense of quadratic constraint builder object.

Example

```
# Retrieve the constraint sense of quadratic constraint builder object.
qconsense = qconstrbuilder.getSense()
```

21.2.17 QConstrBuilderArray Class

To facilitate users to operate on multiple *QConstrBuilder Class* objects, the Python interface of COPT provides QConstrArray object with the following methods:

QConstrBuilderArray()

Synopsis

```
QConstrBuilderArray(qconstrbuilders=None)
```

Description

Create a *QConstrBuilderArray Class* object.

If parameter `qconstrbuilders` is `None`, then create an empty *QConstrBuilderArray Class* object, otherwise initialize the newly created *QConstrBuilderArray Class* object by parameter `qconstrbuilders`.

Arguments

`qconstrbuilders`

Quadratic constraint builder to be added. Optional, `None` by default. It can be *QConstrBuilder Class* object, *QConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty QConstrBuilderArray object.
qconbuilderarr = QConstrBuilderArray()
# Create a QConstrBuilderArray object and initialize it with builders: qconbuilderx and␣
↪qconbuildery
qconbuilderarr = QConstrBuilderArray([qconbuilderx, qconbuildery])
```

QConstrBuilderArray.pushBack()

Synopsis

```
pushBack(qconstrbuilder)
```

Description

Add single or multiple *QConstrBuilder Class* objects.

Arguments

`qconstrbuilder`

Builder of quadratic constraint to be added, which can be *QConstrBuilder Class* object, *QConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add quadratic constraint builder qconbuilderx to qconbuilderarr
qconbuilderarr.pushBack(qconbuilderx)
# Add quadratic constraint builders qconbuilderx and qconbuildery to qconbuilderarr
qconbuilderarr.pushBack([qconbuilderx, qconbuildery])
```

QConstrBuilderArray.getBuilder()

Synopsis

```
getBuilder(idx)
```

Description

Retrieve the corresponding builder object according to the subscript of quadratic constraint builder in *QConstrBuilderArray Class* object.

Arguments

`idx`

Subscript of the quadratic constraint builder in the *QConstrBuilderArray Class* object, starting with 0.

Example

```
# Retrieve the builder with subscript 1 in qconbuilderarr
qconbuilder = qconbuilderarr.getBuilder(1)
```

QConstrBuilderArray.getSize()

Synopsis

```
getSize()
```

Description

Get the number of elements in *QConstrBuilderArray Class* object.

Example

```
# Get the number of builders in qconbuilderarr
qarrsize = qconbuilderarr.getSize()
```

21.2.18 PsdConstraint Class

PsdConstraint object contains related operations of COPT positive semi-definite constraints and provides the following methods:

PsdConstraint.getName()

Synopsis

```
getName()
```

Description

Retrieve the name of positive semi-definite constraint.

Example

```
# Retrieve the name of positive semi-definite constraint 'con'.
conname = con.getName()
```

PsdConstraint.getIdx()

Synopsis

```
getIdx()
```

Description

Retrieve the subscript of positive semi-definite constraint in the model.

Example

```
# Retrieve the subscript of positive semi-definite constraint con.
conidx = con.getIdx()
```

PsdConstraint.setName()

Synopsis

```
setName(newname)
```

Description

Set the name of positive semi-definite constraint.

Arguments

`newname`

The name of positive semi-definite constraint to be set.

Example

```
# Set the name of positive semi-definite constraint 'con'.
con.setName('con')
```

PsdConstraint.getInfo()

Synopsis

`getInfo(infoname)`

Description

Retrieve specified information. Return a constant.

Arguments

`infoname`

Name of the information to be obtained. Please refer to *Information* section for possible values.

Example

```
# Get the lower bound of positive semi-definite constraint con
conlb = con.getInfo(COPT.Info.LB)
```

PsdConstraint.setInfo()

Synopsis

`setInfo(infoname, newval)`

Description

Set new information value to the specified positive semi-definite constraint.

Arguments

`infoname`

The name of the information to be set. Please refer to *Information* section for possible values.

`newval`

New information value to be set.

Example

```
# Set the lower bound of positive semi-definite constraint con
con.setInfo(COPT.Info.LB, 1.0)
```

PsdConstraint.remove()

Synopsis

`remove()`

Description

Delete the positive semi-definite constraint from model.

Example

```
# Delete the positive semi-definite constraint 'conx'
conx.remove()
```

21.2.19 PsdConstrArray Class

To facilitate users to operate on multiple *PsdConstraint Class* objects, the Python interface of COPT provides PsdConstrArray class with the following methods:

PsdConstrArray()

Synopsis

```
PsdConstrArray(constrs=None)
```

Description

Create a *PsdConstrArray Class* object.

If parameter `constrs` is `None`, the create an empty *PsdConstrArray Class* object, otherwise initialize the newly created *PsdConstrArray Class* object with parameter `constrs`

Arguments

`constrs`

Positive semi-definite constraints to be added. `None` by default.

`constrs` can be *PsdConstraint Class* object, *PsdConstrArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty PsdConstrArray object
conarr = PsdConstrArray()
# Create an PsdConstrArray object containing positive semi-definite constraint conx and cony
conarr = PsdConstrArray([conx, cony])
```

PsdConstrArray.pushBack()

Synopsis

```
pushBack(constr)
```

Description

Add single or multiple *PsdConstraint Class* objects.

Arguments

`constr`

Positive semi-definite constraints to be applied. `constrs` can be *PsdConstraint Class* object, *PsdConstrArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add positive semi-definite constraint r to conarr
conarr.pushBack(r)
# Add positive semi-definite constraint r0 and r1 to conarr
conarr.pushBack([r0, r1])
```

PsdConstrArray.getPsdConstr()

Synopsis

```
getPsdConstr(idx)
```

Description

Retrieve the positive semi-definite constraint according to its subscript in *PsdConstrArray Class* object. Return a *PsdConstraint Class* object.

Arguments

`idx`

Subscript of the desired positive semi-definite constraint in *PsdConstrArray Class* object, starting with 0.

Example

```
# Retrieve the positive semi-definite constraint with subscript 1  in conarr
con = conarr.getPsdConstr(1)
```

PsdConstrArray.getSize()

Synopsis

```
getSize()
```

Description

Get the number of elements in *PsdConstrArray Class* object.

Example

```
# Get the number of positive semi-definite constraints in conarr
arrsize = conarr.getSize()
```

21.2.20 PsdConstrBuilder Class

PsdConstrBuilder object contains operations related to temporary constraints when building positive semi-definite constraints, and provides the following methods:

PsdConstrBuilder()

Synopsis

```
PsdConstrBuilder()
```

Description

Create an empty *PsdConstrBuilder Class* object

Example

```
# Create an empty positive semi-definite constraint builder
constrbuilder = PsdConstrBuilder()
```

PsdConstrBuilder.setBuilder()

Synopsis

`setBuilder(expr, sense, rhs)`

Description

Set expression, constraint type and right hand side for positive semi-definite constraint builder.

Arguments

`expr`

The expression to be set, which can be *PsdVar Class* expression or *PsdExpr Class* expression.

`sense`

Sense of constraint. The full list of available tyeps can be found in *Constraint type* section.

`rhs`

The right hand side of constraint.

Example

```
# Set the expresson of positive semi-definite constraint builder as: x + y == 1, and sense of␣
↪constraint as equal
constrbuilder.setBuilder(x + y, COPT.EQUAL, 1)
```

PsdConstrBuilder.setRange()

Synopsis

`setRange(expr, range)`

Description

Set a range positive semi-definite constraint builder where `expr` is less than or equals to 0 and greater than or equals to - `range`.

Arguments

`expr`

The expression to be set, which can be *PsdVar Class* expression or *PsdExpr Class* expression.

`range`

Range of constraint, nonnegative constant.

Example

```
# Set a range positive semi-definite constraint builder: -1 <= x + y - 1 <= 0
constrbuilder.setRange(x + y - 1, 1)
```

PsdConstrBuilder.getPsdExpr()

Synopsis

getPsdExpr()

Description

Retrieve the expression of a positive semi-definite constraint builder object.

Example

```
#  Retrieve the expression of a positive semi-definite constraint builder
psdexpr = constrbuilder.getPsdExpr()
```

PsdConstrBuilder.getSense()

Synopsis

getSense()

Description

Retrieve the constraint sense of positive semi-definite constraint builder object.

Example

```
# Retrieve the constraint sense of positive semi-definite constraint builder object.
consense = constrbuilder.getSense()
```

PsdConstrBuilder.getRange()

Synopsis

getRange()

Description

Retrieve the range of positive semi-definite constraint builder object, i.e. length from lower bound to upper bound of the constraint

Example

```
# Retrieve the range of positive semi-definite constraint builder object
rngval = constrbuilder.getRange()
```

21.2.21 PsdConstrBuilderArray Class

To facilitate users to operate on multiple *PsdConstrBuilder Class* objects, the Python interface of COPT provides PsdConstrBuilderArray object with the following methods:

PsdConstrBuilderArray()

Synopsis

```
PsdConstrBuilderArray(builders=None)
```

Description

Create a *PsdConstrBuilderArray Class* object.

If parameter `builders` is `None`, then create an empty *PsdConstrBuilderArray Class* object, otherwise initialize the newly created *PsdConstrBuilderArray Class* object by parameter `builders`.

Arguments

`builders`

Positive semi-definite constraint builder to be added. Optional, `None` by default. It can be *PsdConstrBuilder Class* object, *PsdConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty PsdConstrBuilderArray object.
conbuilderarr = PsdConstrBuilderArray()
# Create a PsdConstrBuilderArray object containing builders: conbuilderx and conbuildery
conbuilderarr = PsdConstrBuilderArray([conbuilderx, conbuildery])
```

PsdConstrBuilderArray.pushBack()

Synopsis

```
pushBack(builder)
```

Description

Add single or multiple *PsdConstrBuilder Class* objects.

Arguments

`builder`

Builder of positive semi-definite constraint to be added, which can be *PsdConstrBuilder Class* object, *PsdConstrBuilderArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add positive semi-definite constraint builder conbuilderx to conbuilderarr
conbuilderarr.pushBack(conbuilderx)
# Add positive semi-definite constraint builders conbuilderx and conbuildery to conbuilderarr
conbuilderarr.pushBack([conbuilderx, conbuildery])
```

PsdConstrBuilderArray.getBuilder()

Synopsis

```
getBuilder(idx)
```

Description

Retrieve the corresponding builder object according to the subscript of positive semi-definite constraint builder in *PsdConstrBuilderArray Class* object.

Arguments

`idx`

Subscript of the positive semi-definite constraint builder in the *PsdConstrBuilderArray Class* object, starting with 0.

Example

```
# Retrieve the builder with subscript 1 in conbuilderarr
conbuilder = conbuilderarr.getBuilder(1)
```

PsdConstrBuilderArray.getSize()

Synopsis

```
getSize()
```

Description

Get the number of elements in *PsdConstrBuilderArray Class* object.

Example

```
# Get the number of builders in conbuilderarr
arrsize = conbuilderarr.getSize()
```

21.2.22 SOS Class

SOS object contains related operations of COPT SOS constraints. The following methods are provided:

SOS.getIdx()

Synopsis

```
getIdx()
```

Description

Retrieve the subscript of SOS constraint in model.

Example

```
# Retrieve the subscript of SOS constraint sosx.
sosidx = sosx.getIdx()
```

SOS.remove()

Synopsis

`remove()`

Description

Delete the SOS constraint from model.

Example

```
# Delete the SOS constraint 'sosx'
sosx.remove()
```

21.2.23 SOSArray Class

To facilitate users to operate on a set of *SOS Class* objects, COPT designed SOSArray class in Python interface. The following methods are provided:

SOSArray()

Synopsis

`SOSArray(soss=None)`

Description

Create a *SOSArray Class* object.

If parameter `soss` is `None`, then build an empty *SOSArray Class* object, otherwise initialize the newly created *SOSArray Class* object with `soss`.

Arguments

`soss`

SOS constraint to be added. Optional, `None` by default. It can be *SOS Class* object, *SOSArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create a new SOSArray object
sosarr = SOSArray()
# Create a SOSArray object, and initialize it with SOS constraints sosx and sosy.
sosarr = SOSArray([sosx, sosy])
```

SOSArray.pushBack()

Synopsis

`pushBack(sos)`

Description

Add one or multiple *SOS Class* objects.

Arguments

`sos`

SOS constraints to be added, which can be *SOS Class* object, *SOSArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add SOS constraint sosx to sosarr
sosarr.pushBack(sosx)
# Add SOS constraints sosx and sosy to sosarr
sosarr.pushBack([sosx, sosy])
```

SOSArray.getSOS()

Synopsis

```
getSOS(idx)
```

Description

Retrieve the corresponding SOS constraint according to its subscript in *SOSArray Class* object and return a *SOS Class* object.

Arguments

`idx`

Indice of the SOS constraint in *SOSArray Class* object, starting with 0.

Example

```
# Retrieve the SOS constraint with indice of 1 in sosarr
sos = sosarr.getSOS(1)
```

SOSArray.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of elements in *SOSArray Class* object.

Example

```
# Retrieve the number of SOS constraints in sosarr.
arrsize = sosarr.getSize()
```

21.2.24 SOSBuilder Class

For easy access builders of SOS constraints, SOSBuilder class provides the following methods:

SOSBuilder()

Synopsis

```
SOSBuilder()
```

Description

Create an empty *SOSBuilder Class* object.

Example

```
# Create an empty SOSBuilder object.
sosbuilder = SOSBuilder()
```

SOSBuilder.setBuilder()

Synopsis

```
setBuilder(sostype, vars, weights=None)
```

Description

Set type, variable, weight of variable on *SOSBuilder Class* object.

Arguments

`sostype`

SOS constraint type. Full list of available types can be found in SOS-constraint types.

`vars`

Variables of SOS constarint, which can be *VarArray Class* object, list, dictionary or *tupledict Class* object.

`weights` Weights of variables in SOS constraint. Optional, `None` by default. Could be list, dictionary or *tupledict Class* object.

Example

```
# Set the type of SOS constraint builder as SOS1, variables x and y, weights of variables as 1␣
↪and 2 respectively.
sosbuilder.setBuilder(COPT.SOS_TYPE1, [x, y], [1, 2])
```

SOSBuilder.getType()

Synopsis

```
getType()
```

Description

Retrieve the SOS constraint type of *SOSBuilder Class* object.

Example

```
# Retrieve the type of SOS constraint builder sosx.
sostype = sosbuilder.getType(sosx)
```

SOSBuilder.getVar()

Synopsis

```
getVar(idx)
```

Description

Retrieve the corresponding variables according to its indice in *SOSBuilder Class* object, and return a *Var Class* object.

Arguments

`idx`

Indice of the variable in *SOSBuilder Class* object, starting with 0.

Example

```
# Retrieve the variable in SOS constraint builder sosx with indice of 1
sosvar = sosx.getVar(1)
```

SOSBuilder.getVars()

Synopsis

```
getVars()
```

Description

Retrieve all variables in *SOSBuilder Class* objects, and return a *VarArray Class* object.

Example

```
# Retrieve all variables in SOS constraint builder sosx.
sosvars = sosx.getVars()
```

SOSBuilder.getWeight()

Synopsis

```
getWeight(idx)
```

Description

Retrieve the corresponding weight of variable according to its indice in *SOSBuilder Class* object.

Arguments

`idx`

Indice of the variable in *SOSBuilder Class* object, starting with 0.

Example

```
# Retrieve the corresponding weight of variable according to its indice in the SOS constraint␣
↪builder sosx.
sosweight = sosx.getWeight(1)
```

SOSBuilder.getWeights()

Synopsis

```
getWeights()
```

Description

Retrieve weights of all the variables in *SOSBuilder Class* object.

Example

```
# Retrieve weights of all the variables in SOS constraint builder sosx.
sosweights = sosx.getWeights()
```

SOSBuilder.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of elements in *SOSBuilder Class* object.

Example

```
# Retrieve the number of elements in SOS constraint builder sosx.
sossize = sosx.getSize()
```

21.2.25 SOSBuilderArray Class

In order to facilitate users to operate on a set of *SOSBuilder Class* objects, COPT provides SOSBuilderArray class in Python interface, providing the following methods:

SOSBuilderArray()

Synopsis

```
SOSBuilderArray(sosbuilders=None)
```

Description

Create a *SOSBuilderArray Class* object.

If parameter `sosbuilders` is `None`, then create an empty *SOSBuilderArray Class* object, otherwise initialize the newly created *SOSBuilderArray Class* object with parameter `sosbuilders`.

Arguments

`sosbuilders`

SOS constraint builder to be added. Optional, `None` by default. Could be *SOSBuilder Class* object, *SOSBuilderArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty SOSBuilderArray object.
sosbuilderarr = SOSBuilderArray()
# Create a SOSBuilderArray object and initialize it with SOS constraint builder sosx and sosy
sosbuilderarr = SOSBuilderArray([sosx, sosy])
```

SOSBuilderArray.pushBack()

Synopsis

```
pushBack(sosbuilder)
```

Description

Add one or multiple *SOSBuilder Class* objects.

Arguments

`sosbuilder`

SOS constraint builderto be added. Cound be *SOSBuilder Class* object, *SOSBuilderArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add SOS constraint builder sosx to sosbuilderarr
sosbuilderarr.pushBack(sosx)
```

SOSBuilderArray.getBuilder()

Synopsis

```
getBuilder(idx)
```

Description

Retrieve the corresponding builder according to the indice of SOS constraint builder in *SOSBuilderArray Class* object.

Arguments

`idx`

Indice of the SOS constraint builder in *SOSBuilderArray Class* object, starting with 0.

Example

```
# Retrieve the SOS constraint builder with indice of 1 in sosbuilderarr
sosbuilder = sosbuilderarr.getBuilder(1)
```

SOSBuilderArray.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of elements in *SOSBuilderArray Class* object.

Example

```
# Retrieve the number of elements in sosbuilderarr
sosbuildersize = sosbuilderarr.getSize()
```

21.2.26 Cone Class

Cone object contains related operations of COPT Second-Order-Cone (SOC) constraints. The following methods are provided:

Cone.getIdx()

Synopsis

```
getIdx()
```

Description

Retrieve the subscript of SOC constraint in model.

Example

```
# Retrieve the subscript of SOC constraint cone.
coneidx = cone.getIdx()
```

Cone.remove()

Synopsis

```
remove()
```

Description

Delete the SOC constraint from model.

Example

```
# Delete the SOC constraint 'cone'
cone.remove()
```

21.2.27 ConeArray Class

To facilitate users to operate on a set of *Cone Class* objects, COPT designed ConeArray class in Python interface. The following methods are provided:

ConeArray()

Synopsis

```
ConeArray(cones=None)
```

Description

Create a *ConeArray Class* object.

If parameter `cones` is `None`, then build an empty *ConeArray Class* object, otherwise initialize the newly created *ConeArray Class* object with `cones`.

Arguments

`cones`

Second-Order-Cone constraint to be added. Optional, `None` by default. It can be *Cone Class* object, *ConeArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create a new ConeArray object
conearr = ConeArray()
# Create a ConeArray object, and initialize it with SOC constraints conex and coney.
conearr = ConeArray([conex, coney])
```

ConeArray.pushBack()

Synopsis

```
pushBack(cone)
```

Description

Add one or multiple *Cone Class* objects.

Arguments

`cone`

Second-Order-Cone constraints to be added, which can be *Cone Class* object, *ConeArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add SOC constraint conex to conearr
conearr.pushBack(conex)
# Add SOC constraints conex and coney to conearr
conearr.pushBack([conex, coney])
```

ConeArray.getCone()

Synopsis

```
getCone(idx)
```

Description

Retrieve the corresponding Second-Order-Cone (SOC) constraint according to its subscript in *ConeArray Class* object and return a *Cone Class* object.

Arguments

`idx`

Indice of the SOC constraint in *ConeArray Class* object, starting with 0.

Example

```
# Retrieve the SOC constraint with indice of 1 in conearr
cone = conearr.getCone(1)
```

ConeArray.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of elements in *ConeArray Class* object.

Example

```
# Retrieve the number of SOC constraints in conearr.
arrsize = conearr.getSize()
```

21.2.28 ConeBuilder Class

For easy access builders of Second-Order-Cone (SOC) constraints, ConeBuilder class provides the following methods:

ConeBuilder()

Synopsis

```
ConeBuilder()
```

Description

Create an empty *ConeBuilder Class* object.

Example

```
# Create an empty ConeBuilder object.
conebuilder = ConeBuilder()
```

ConeBuilder.setBuilder()

Synopsis

```
setBuilder(conetype, vars)
```

Description

Set type, variables of *ConeBuilder Class* object.

Arguments

`conetype`

Type of Second-Order-Cone (SOC) constraint. Full list of available types can be found in SOC-constraint types.

`vars`

Variables of SOC constarint, which can be *VarArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Set type as regular, variables as [z, x, y] for SOC constraint builder
conebuilder.setBuilder(COPT.CONE_QUAD, [z, x, y])
```

ConeBuilder.getType()

Synopsis

```
getType()
```

Description

Retrieve the Second-Order-Cone (SOC) constraint type of *ConeBuilder Class* object.

Example

```
# Retrieve the type of SOC constraint builder conex.
conetype = conebuilder.getType(conex)
```

ConeBuilder.getVar()

Synopsis

```
getVar(idx)
```

Description

Retrieve the corresponding variables according to its indice in *ConeBuilder Class* object, and return a *Var Class* object.

Arguments

`idx`

Indice of the variable in *ConeBuilder Class* object, starting with 0.

Example

```
# Retrieve the variable in SOC constraint builder conex with indice of 1
conevar = conex.getVar(1)
```

ConeBuilder.getVars()

Synopsis

```
getVars()
```

Description

Retrieve all variables in *ConeBuilder Class* objects, and return a *VarArray Class* object.

Example

```
# Retrieve all variables in SOC constraint builder conex.
conevars = conex.getVars()
```

ConeBuilder.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of elements in *ConeBuilder Class* object.

Example

```
# Retrieve the number of elements in SOC constraint builder conex.
conesize = conex.getSize()
```

21.2.29 ConeBuilderArray Class

In order to facilitate users to operate on a set of *ConeBuilder Class* objects, COPT provides ConeBuilderArray class in Python interface, providing the following methods:

ConeBuilderArray()

Synopsis

```
ConeBuilderArray(conebuilders=None)
```

Description

Create a *ConeBuilderArray Class* object.

If parameter `conebuilders` is `None`, then create an empty *ConeBuilderArray Class* object, otherwise initialize the newly created *ConeBuilderArray Class* object with parameter `conebuilders`.

Arguments

`conebuilders`

SOC constraint builder to be added. Optional, `None` by default. Could be *ConeBuilder Class* object, *ConeBuilderArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create an empty ConeBuilderArray object.
conebuilderarr = ConeBuilderArray()
# Create a ConeBuilderArray object and initialize it with SOC constraint builder conex and␣
↪coney
conebuilderarr = ConeBuilderArray([conex, coney])
```

ConeBuilderArray.pushBack()

Synopsis

`pushBack(coneuilder)`

Description

Add one or multiple *ConeBuilder Class* objects.

Arguments

`conebuilder`

SOC constraint builder to be added. Cound be *ConeBuilder Class* object, *ConeBuilderArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Add SOC constraint builder conex to conebuilderarr
conebuilderarr.pushBack(conex)
```

ConeBuilderArray.getBuilder()

Synopsis

`getBuilder(idx)`

Description

Retrieve the corresponding builder according to the indice of SOC constraint builder in *ConeBuilderArray Class* object.

Arguments

`idx`

Indice of the SOC constraint builder in *ConeBuilderArray Class* object, starting with 0.

Example

```
# Retrieve the SOC constraint builder with indice of 1 in conebuilderarr
conebuilder = conebuilderarr.getBuilder(1)
```

ConeBuilderArray.getSize()

Synopsis

`getSize()`

Description

Retrieve the number of elements in *ConeBuilderArray Class* object.

Example

```
# Retrieve the number of elements in conebuilderarr
conebuildersize = conebuilderarr.getSize()
```

21.2.30 GenConstr Class

For easy access to Indicator constraints, COPT provides GenConstr class which containing the following methods:

GenConstr.getIdx()

Synopsis

`getIdx()`

Description

Retrieve the subscript of Indicator constraint in model.

Example

```
# Retrieve the indice of Indicator constraint indicx
indidx = indicx.getIdx()
```

GenConstr.remove()

Synopsis

`remove()`

Description

Delete the indicator constraint from model.

Example

```
# Delete indicator constraint 'indx'
indx.remove()
```

21.2.31 GenConstrArray Class

In order to facilitate users to operate on a set of *GenConstr Class* objects, COPT provides GenConstrArray class in Python interface, providing the following methods:

GenConstrArray()

Synopsis

`GenConstrArray(genconstrs=None)`

Description

Create a *GenConstrArray Class* object.

If parameter `genconstrs` is `None`, then create an empty *GenConstrArray Class* object, otherwise initialize the newly created *GenConstrArray Class* object with parameter `genconstrs`.

Arguments

`genconstrs`

Indicator constraint to be added. Optional, `None` by default. Could be *GenConstr Class* object, *GenConstrArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Create a new GenConstrArray object
genconstrarr = GenConstrArray()
# Create a GenConstrArray object and user Indicator constraints genx and geny to initialize it.
genconstrarr = GenConstrArray([genx, geny])
```

GenConstrArray.pushBack()

Synopsis

`pushBack(genconstr)`

Description

Add one or multiple *GenConstr Class* objects.

Arguments

`constrs` The Indicator constraint to be added. Cound be *GenConstr Class* object, *GenConstrArray Class* object, list, dictionary or *tupledict Class* object.

Example

```
# Aff Indicator constraint genx to genconarr
genconarr.pushBack(genx)
# Add Indicator constraint genx and geny to genconarr
genconarr.pushBack([genx, geny])
```

GenConstrArray.getGenConstr()

Synopsis

`getGenConstr(idx)`

Description

Retrieve the corresponding Indicator constraint according to the indice of Indicator constraint in *GenConstrArray Class* object, and return a *GenConstr Class* object.

Arguments

`idx`

Indice of the Indicator constraint in *GenConstrArray Class*, starting with 0.

Example

```
# Retrieve the Indicator constraint with indice of 1 in genconarr
genconstr = genconarr.getGenConstr(1)
```

GenConstrArray.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of elements in *GenConstrArray Class* object.

Example

```
# Retrieve the number of elements in genconarr
genconsize = genconarr.getSize()
```

21.2.32 GenConstrBuilder Class

GenConstrBuilder object contains operations for building Indicator constraints, and provides the following methods:

GenConstrBuilder()

Synopsis

```
GenConstrBuilder()
```

Description

Create an empty *GenConstrBuilder Class* object.

Example

```
# Create an empty GenConstrBuilder object
genconbuilder = GenConstrBuilder()
```

GenConstrBuilder.setBuilder()

Synopsis

```
setBuilder(var, val, expr, sense)
```

Description

Set Indicator variable, the value of Indicator variable, the expression and constraint sense of a *GenConstrBuilder Class* object.

Arguments

`var`

Indicator variable.

`val`

Value of an Indicator variable.

`expr`

Expression of linear constraint, which can be *Var Class* object or *LinExpr Class* object.

`sense`

Sense for the linear constraint. Please refer to *Constraint type* for possible values.

Example

```
# Set Indicator variable of Indicator constraint builder to x. When x is true, the linear␣
↪constraint x + y == 1 holds
genconbuilder.setBuilder(x, True, x + y - 1, COPT.EQUAL)
```

GenConstrBuilder.getBinVar()

Synopsis

getBinVar()

Description

Retrieve the Indicator variable of a *GenConstrBuilder Class* object.

Example

```
# Retrieve the Indicator variable of Indicator constraint builder genbuilderx
indvar = genbuilderx.getBinVar()
```

GenConstrBuilder.getBinVal()

Synopsis

getBinVal()

Description

Retrieve the value of Indicator variable of a *GenConstrBuilder Class* object.

Example

```
# Retrieve the value when the Indicator variable of Indicator constraint builder genbuilderx␣
↪is valid
indval = genbuilderx.getBinVal()
```

GenConstrBuilder.getExpr()

Synopsis

getExpr()

Description

Retrieve the linear expression of a *GenConstrBuilder Class* object.

Example

```
# Retrieve the linear expression of Indicator constraint builder genbuilderx
linexpr = genbuilderx.getExpr()
```

GenConstrBuilder.getSense()

Synopsis

```
getSense()
```

Description

Retrieve the sense for the linear constraint of a *GenConstrBuilder Class* object.

Example

```
# Retrieve the sense for the linear constraint of Indicator constraint builder genbuilderx
linsense = genbuilderx.getSense()
```

21.2.33 GenConstrBuilderArray Class

To facilitate users to operate on multiple *GenConstrBuilder Class* objects, the Python interface of COPT provides GenConstrBuilderArray object with the following methods:

GenConstrBuilderArray()

Synopsis

```
GenConstrBuilderArray(genconstrbuilders=None)
```

Description

Create a *GenConstrBuilderArray Class* object.

If argument `genconstrbuilders` is `None`, then create an empty *GenConstrBuilderArray Class* object; otherwise use the argument `genconstrbuilders` to initialize the newly created *GenConstrBuilderArray Class* object.

Arguments

`genconstrbuilders`

Indicator constraint builder to add. Optional, `None` by default. It can be *GenConstrBuilder Class* object, *GenConstrBuilderArray Class* object, list, dict, or *tupledict Class* object.

Example

```
# Create an empty GenConstrBuilderArray object
genbuilderarr = GenConstrBuilderArray()
# Create a GenConstrBuilderArray object and use Indicator constraint builder genbuilderx and␣
↪genbuildery to initialize it.
genbuilderarr = GenConstrBuilderArray([genbuilderx, genbuildery])
```

GenConstrBuilderArray.pushBack()

Synopsis

```
pushBack(genconstrbuilder)
```

Description

Add single or multiple *GenConstrBuilder Class* objects.

Arguments

`genconstrbuilder`

Indicator constraint builders to add, which can be *GenConstrBuilder Class* object, *GenConstrBuilderArray Class* object, list, dict, or *tupledict Class* object.

Example

```
# Add an Indicator constraint builder to genbuilderarr
genbuilderarr.pushBack(genbuilderx)
# Add Indicator constraint builders genbuilderx and genbuildery to genbuilderarr
genbuilderarr.pushBack([genbuilderx, genbuildery])
```

GenConstrBuilderArray.getBuilder()

Synopsis

`getBuilder(idx)`

Description

Retrieve the Indicator constraint builder according to its index in the *GenConstrBuilderArray Class* object, and return a *GenConstrBuilder Class* object.

Arguments

`idx`

Index of the Indicator constraint builder in the *GenConstrBuilderArray Class* object, starting with 0.

Example

```
# Retrieve the Indicator constraint builder whose index in genbuilderarr is 1
genbuilder = genbuilderarr.getBuilder(1)
```

GenConstrBuilderArray.getSize()

Synopsis

`getSize()`

Description

Retrieve the number of elements in the *GenConstrBuilderArray Class* object.

Example

```
# Retrieve the number of elements in genbuilderarr
genbuildersize = genbuilderarr.getSize()
```

21.2.34 Column Class

To facilitate users to model by column, the Python interface of COPT provides Column object with the following methods:

Column()

Synopsis

```
Column(constrs=0.0, coeffs=None)
```

Description

Create an *Column Class* object.

If argument `constrs` is `None` and argument `coeffs` is `None`, then create an empty *Column Class* object; otherwise use the argument `constrs` and `coeffs` to initialize the newly created *Column Class* object. If argument `constrs` is a *Constraint Class* or *Column Class* object, then argument `coeffs` is a constant. If argument `coeffs` is `None`, then it is considered to be constant 1.0; If argument `constrs` is a list and argument `coeffs` is `None`, then the elements of argument `constrs` are constraint-coefficient pairs; For other forms of arguments, call method `addTerms` to initialize the newly created *Column Class* object.

Arguments

`constrs`

Linear constraint.

`coeffs`

Coefficient for variables in the linear constraint.

Example

```
# Create an empty Column object
col = Column()
# Create a Column object and add two terms: coefficient is 2 in constraint conx and 3 in␣
↪constraint cony
col = Column([(conx, 2), (cony, 3)])
# Create a Column object and add two terms: coefficient is 1 in constraint conxx and 2 in␣
↪constraint conyy
col = Column([conxx, conyy], [1, 2])
```

Column.getCoeff()

Synopsis

```
getCoeff(idx)
```

Description

Retrieve the coefficient according to its index in the *Column Class* object.

Arguments

`idx`

Index for the element, starting with 0.

Example

```
# Retrieve the coefficient whose index is 0 in col
coeff = col.getCoeff(0)
```

Column.getConstr()

Synopsis

```
getConstr(idx)
```

Description

Retrieve the linear constraint according to its index in the *Column Class* object.

Arguments

`idx`

Index for the element, starting with 0.

Example

```
# Retrieve the linear constraint whose index is 1 in col
constr = col.getConstr(1)
```

Column.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of elements in the *Column Class* object.

Example

```
# Retrieve the number of elements in col
colsize = col.getSize()
```

Column.addTerm()

Synopsis

```
addTerm(constr, coeff=1.0)
```

Description

Add a new term.

Arguments

`constr`

The linear constraint for the term to add.

`coeff`

The coefficient for the term to add. Optional, 1.0 by default.

Example

```
# Add an term to col, whose constraint is cony and coefficient is 2.0
col.addTerm(cony, 2.0)
# Add an term to col, whose constraint is conx and coefficient is 1.0
col.addTerm(conx)
```

Column.addTerms()

Synopsis

```
addTerms(constrs, coeffs)
```

Description

Add single or multiple terms.

If argument `constrs` is *Constraint Class* object, then argument `coeffs` is constant; If argument `constrs` is *ConstrArray Class* object or list, then argument `coeffs` is constant or list; If argument `constrs` is dictionary or *tupledict Class* object, then argument `coeffs` is constant, dict, or *tupledict Class* object.

Arguments

`constrs`

The linear constraints for terms to add.

`coeffs`

The coefficients for terms to add.

Example

```
# Add two terms: constraint conx with coefficient 2.0, constraint cony with coefficient 3.0
col.addTerms([conx, cony], [2.0, 3.0])
```

Column.addColumn()

Synopsis

```
addColumn(col, mult=1.0)
```

Description

Add a new column to current column.

Arguments

`col`

Column to add.

`mult`

Magnification coefficient for added column. Optional, 1.0 by default.

Example

```
# Add column coly to column colx. The magnification coefficient for coly is 2.0
colx.addColumn(coly, 2.0)
```

Column.clone()

Synopsis

```
clone()
```

Description

Create a deep copy of a column.

Example

```
# Create a deep copy of column col
colcopy = col.clone()
```

Column.remove()

Synopsis

```
remove(item)
```

Description

Remove a term from a column.

If argument `item` is a constant, then remove the term by its index; otherwise argument `item` is a *Constraint Class* object.

Arguments

`item`

Constant index or the linear constraint for the term to be removed.

Example

```
# Remove the term whose index is 2 from column col
col.remove(2)
# Remove the term of the linear constraint conx from col
col.remove(conx)
```

Column.clear()

Synopsis

```
clear()
```

Description

Remove all terms from a column.

Example

```
# Remove all terms from column col
col.clear()
```

21.2.35 ColumnArray Class

To facilitate users to operate on multiple *Column Class* objects, the Python interface of COPT provides ColumnArray object with the following methods:

ColumnArray()

Synopsis

```
ColumnArray(columns=None)
```

Description

Create a *ColumnArray Class* object.

If argument `columns` is `None`, then create an empty *ColumnArray Class* object; otherwise use argument `columns` to initialize the newly created *ColumnArray Class* object.

Arguments

`columns`

Columns to add. Optional, `None` by default. It can be *Column Class* object, *ColumnArray Class* object, list, dict, or *tupledict Class* object.

Example

```
# Create an empty ColumnArray object
colarr = ColumnArray()
# Create a ColumnArray object and use columns colx and coly to initialize it
colarr = ColumnArray([colx, coly])
```

ColumnArray.pushBack()

Synopsis

`pushBack(column)`

Description

Add single or multiple *Column Class* objects.

Arguments

`column`

Columns to add, which can be *Column Class* object, *ColumnArray Class* object, list, dict, or *tupledict Class* object.

Example

```
# Add column colx to colarr
colarr.pushBack(colx)
# Add columns colx and coly to colarr
colarr.pushBack([colx, coly])
```

ColumnArray.getColumn()

Synopsis

`getColumn(idx)`

Description

Retrieve the column according to its index in a *ColumnArray Class* object. Return a *Column Class* object.

Arguments

`idx`

Index of the column in the *ColumnArray Class* object, starting with 0.

Example

```
# Retrieve the column whose index is 1 in colarr
col = colarr.getColumn(1)
```

ColumnArray.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of elements in a *ColumnArray Class* object.

Example

```
# Retrieve the number of element in colarr
colsize = colarr.getSize()
```

ColumnArray.clear()

Synopsis

```
clear()
```

Description

Remove all terms from a *ColumnArray Class* object.

Example

```
# Remove all terms from colarr
colarr.clear()
```

21.2.36 MVar Class

The MVar class is used in COPT to build multi-dimensional variables and supports NumPy's multi-dimensional array operations. We recommend to generate it through the method `addVars` or `addMVar` of the model class, although it can also be converted and generated through the two built-in class methods `fromlist` and `fromvar` . The following member methods are provided:

MVar.fromlist()

Synopsis

```
fromlist(vars)
```

Description

Generate a *MVar Class* object from a set of *Var Class* objects. This is the class generation method and can be called directly without MVar object.

Arguments

`vars`

A set of Var objects, which can be a multi-dimensional list or ndarray.

Return value

new MVar object whose dimensions depend on the dimensions of the arguments vars.

Example

```
vars = model.addVars(4)
mx_1d = MVar.fromlist(vars)
mx_2d = MVar.fromlist([vars[0], vars[1]], [vars[2], vars[3]])
```

MVar.fromvar()

Synopsis

`fromvar(var)`

Description

Generate a 0-dimensional *MVar Class* object from a *Var Class* object. This is the class generation method and can be called directly without MVar object.

Arguments

`var`

A Var object.

Return value

The new 0-dimensional MVar object.

Example

```
x = model.addVar()
mx_0d = MVar.fromvar(x)
```

MVar.clone()

Synopsis

`clone()`

Description

Deep-copy a *MVar Class* object.

Return value

new MVar object

Example

```
# Create a 2-D variable and make a copy. Note that the actual variable is not␣
↪incremented.
mx = model.addMVar((3, 2), nameprefix="mx")
mx_copy = mx.clone()
```

MVar.diagonal()

Synopsis

`diagonal(offset=0, axis1=0, axis2=1)`

Description

Generate a *MVar Class* object whose elements are the elements on the diagonal of the original MVar object.

Arguments

`offset`

Optional parameter, indicating the offset of the diagonal, the default value is 0. If the value is greater than 0, it means the diagonal upward offset; if the value is less than 0, it means the diagonal downward offset.

`axis1`

Optional parameter, the axis to use as the first axis of the 2D sub MVar, from where the diagonal should start. The default first axis is 0.

`axis2`

Optional parameter, the axis to use as the second axis of the 2D sub MVar, from where the diagonal should start. The default second axis is 1.

Return value

new MVar object

Example

```
mx = model.addMVar((5, 5), nameprefix="mx")
diag_m0 = mx.diagonal()
diag_a1 = mx.diagonal(1)
diag_b1 = mx.diagonal(-1)
```

MVar.getAttr()

Synopsis

`getAttr(attrname)`

Description

Get the property value of each variable inside MVar.

Arguments

`attrname`

The name of the property being queried.

Return value

Returns a NumPy ndarray with the same dimension as the MVar object, whose elements are the attribute values of the corresponding variable.

Example

```
mx = model.addMVar(3)
print(mx.getAttr("x"))
```

MVar.item()

Synopsis

`item()`

Description

Get the Var variable inside the 0-dimensional MVar. Raises a ValueError exception if the MVar object is not 0-dimensional.

Return value

Returns the Var object.

Example

```
mx = model.addMVar(3)
var = mx[0].item()
```

MVar.reshape()

Synopsis

```
reshape(shape, order='C')
```

Description

Returns a new MVar object whose elements remain unchanged but whose shape is transformed by the parameter shape.

Arguments

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MVar object.

`order`

Optional parameter, the default is the character 'C', which means it is compatible with the C language, that is, it is stored in rows; it can also be set to the character 'F', that is, it is stored in columns, and it is compatible with the Fortune language.

Return value

Returns a new MVar object with the same elements as the original MVar object but with a different shape.

Example

```
mx = model.addMVar(6)
mx_2x2 = mx.reshape((2, 3))
```

MVar.setAttr()

Synopsis

```
setAttr(attrname, newvalue)
```

Description

Sets the attribute value of each variable inside the MVar.

Arguments

`attrname`

The name of the property being queried.

`newvalue`

The property value to be set.

Example

```
mx = model.addMVar(3)
mx.setAttr("ub", 9.0)
mx.setAttr(COPT.Info.LB, 0.0)
```

MVar.sum()

Synopsis

`sum(axis=None)`

Description

Sum the variables in the MVar, returning a new *MLinExpr Class* object.

Arguments

`axis`

Optional integer parameter, the default value is None, that is, to sum up variables one by one. Otherwise, sum over the given axis.

Return value

Returns an MLinExpr object representing the sum of the corresponding variables.

Example

```
mx = model.addMVar((3, 5))
sum_all = mx.sum() #Return 0-dimensional MLinExpr object
sum_row = mx.sum(axis = 0) #Return a 1-dimensional MLinExpr object with a shape of␣
↪(5, )
```

MVar.tolist()

Synopsis

`tolist()`

Description

Convert an MVar object to a one-dimensional list of Var objects.

Return value

Returns a one-dimensional list containing Var objects.

Example

```
mx = model.addMVar((3, 5))
print(mx.tolist())
```

MVar.transpose()

Synopsis

`transpose()`

Description

Generates a new MVar object that is the transpose of the original MVar object.

Return value

Return the new MVar object.

Example

```
mx = model.addMVar((3, 5))
print(mx.transpose().shape) #its shape is (5, 3)
```

MVar.ndim

Synopsis

`ndim`

Description

Dimensions of the MVar object.

Return value

Integer value.

Example

```
mx = model.addMVar((3, 5))
print(mx.ndim) #ndim = 2
```

MVar.shape

Synopsis

`shape`

Description

The shape of the MVar object.

Return value

Integer tuple.

Example

```
mx = model.addMVar((3,))
print(mx.shape) # shape = (3, )
```

MVar.size

Synopsis

`size`

Description

The number of Var variables in the MVar object.

Return value

Integer value.

Example

```
mx = model.addMVar((3, 4))
print(mx.size) # size = 12
```

MVar.T

Synopsis

```
T
```

Description

Transpose of the MVar object. Similar to the class method transpose().

Return value

Returns the transposed MVar object.

Example

```
mx = model.addMVar((3, 4))
print(mx.T.shape) # shape = (4, 3)
```

21.2.37 MConstr Class

The MConstr class holds multi-dimensional linear constraints in COPT and supports NumPy's multi-dimensional array operations. It is generated by the method `addConstrs` or `addMConstr` of the model class. The following member methods are provided:

MConstr.getAttr()

Synopsis

```
getAttr(attrname)
```

Description

Get the attribute value of each constraint within MConstr.

Arguments

`attrname`

The name of the property being queried.

Return value

Returns a NumPy ndarray with the same dimension as the MConstr object, whose elements are the attribute values of the corresponding constraint.

Example

```
a = np.random.rand(4)
mx = m.addMVar((4, 3), nameprefix="mx")
b = np.random.rand(3)
mc = m.addConstrs(a @ mx <= b)
print(mc.getAttr("pi"))
```

MConstr.item()

Synopsis

```
item()
```

Description

Get the constraint object in 0-dimensional MConstr. If the MConstr object is not 0-dimensional, a ValueError exception is raised.

Return value

Returns the linear constraint object.

Example

```
mc = m.addConstrs(a @ mx <= b)
print(mc[0].item())
```

MConstr.reshape()

Synopsis

```
reshape(shape, order='C')
```

Description

Returns a new MConstr object whose elements remain unchanged but whose shape is transformed by the argument shape.

Arguments

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MConstr object.

`order`

Optional parameter, the default is the character 'C', which means it is compatible with the C language, that is, it is stored in rows; it can also be set to the character 'F', that is, it is stored in columns, and it is compatible with the Fortune language.

Return value

Returns a new MConstr object with the same elements as the original MConstr object but with a different shape.

Example

```
mc = m.addConstrs(a @ mx <= b)
mc_2x2 = mc.reshape((2, 2))
```

MConstr.setAttr()

Synopsis

```
setAttr(attrname, newvalue)
```

Description

Sets property values for each constraint within MConstr.

Arguments

`attrname`

The name of the property being queried.

`newvalue`

The property value to be set.

Example

```
mc = model.addConstrs(a @ mx <= b)
mc.setAttr("obj", 9.0)
```

MConstr.tolist()

Synopsis

```
tolist()
```

Description

Convert the MConstr object to a one-dimensional list of constraints.

Return value

Returns a one-dimensional list containing Constraint objects.

Example

```
mc = m.addConstrs(a @ mx <= b)
print(mc.tolist())
```

MConstr.transpose()

Synopsis

```
transpose()
```

Description

Generates a new MConstr object that is the transpose of the original MConstr object.

Return value

Returns the transposed MConstr object.

Example

```
mc = m.addConstrs(a @ mx <= b)
print(mc.transpose())
```

MConstr.ndim

Synopsis

ndim

Description

Dimensions of the MConstr object.

Return value

Integer value.

Example

```
mc = m.addConstrs(a @ mx <= b)
print(mc.ndim)
```

MConstr.shape

Synopsis

shape

Description

The shape of the MConstr object.

Return value

Integer tuple.

Example

```
mc = m.addConstrs(a @ mx <= b)
print(mc.shape)
```

MConstr.size

Synopsis

size

Description

The number of constraints of the MConstr object.

Return value

Integer value.

Example

```
mc = m.addConstrs(a @ mx <= b)
print(mc.size)
```

MConstr.T

Synopsis

T

Description

Transpose of the MConstr object. Similar to the class method transpose().

Return value

Returns the transposed MConstr object.

Example

```
A = np.ones([2, 4])
mx = m.addMVar((4, 3), nameprefix="mx")
mc = m.addConstrs(A @ X == 0.0)
print(mc.T.shape) # shape = (3, 2)
```

21.2.38 MConstrBuilder Class

The MConstrBuilder class is used to build multi-dimensional linear constraints in COPT and supports NumPy's multi-dimensional array operations. Users might create a MConstrBuilder object by its constructor with a list of *ConstrBuilder Class* objects, or simply by overloaded comparison operator of *MLinExpr Class*. The following member methods are provided:

MConstrBuilder()

Synopsis

MConstrBuilder(args, shape=None)

Description

construtor of MConstrBuilder.

Arguments

args

one or a set of *ConstrBuilder Class* objects, in form of Python list or NumPy ndarray.

shape

an integer, or tuple of integers, which is the shape of new MConstrBuilder object.

Example

```
vars = m.addVars(4)
builders = [x <= 1.0 for x in vars]
mcb = MConstrBuilder(builders, (2, 2))

# or by overloaded comparison operator of MVar
mx = m.addMVar((3, 2))
mcb = mx >= 1.0

# or immediately passed to addConstrs()
model.addConstrs(mx >= 1.0)
```

21.2.39 MQConstrBuilder Class

The MQConstrBuilder class is used to build multi-dimensional quadratic constraints in COPT and supports NumPy's multi-dimensional array operations. Users might create a MQConstrBuilder object by its constructor with a list of *QConstrBuilder Class* objects, or simply by overloaded comparison operator of *MQuadExpr Class*. The following member methods are provided:

MQConstrBuilder()

Synopsis

```
MQConstrBuilder(args, shape=None)
```

Description

construtor of MQConstrBuilder.

Arguments

`args`

one or a set of *QConstrBuilder Class* objects, in form of Python list or NumPy ndarray.

`shape`

an integer, or tuple of integers, which is the shape of new MQConstrBuilder object.

Example

```
x = model.addVar()
mqcb = MQConstrBuilder(x * x <= 9.0)

# or by overloaded comparison operator of MQuadExpr
mx = model.addMVar(3, 3)
mqcb = mx @ mx >= 1.0

# or immediately passed to addConstrs()
ma = model.addMVar(2)
A = np.full((2,3), 1)
mb = model.addMVar(3)
model.addQConstr(ma @ A @ mb <= 1.0)
```

21.2.40 MLinExpr Class

The MLinExpr class is used in COPT to build multi-dimensional linear expressions and supports NumPy's multi-dimensional array operations. The MLinExpr object with an initial value of 0.0 can be generated by the class generation method `zeros()` , or by *MVar Class* object to generate a linear combination. The following member methods are provided:

MLinExpr.zeros()

Synopsis

zeros(shape)

Description

This is the class generation method and can be called directly without the MLinExpr object.

Arguments

shape

The value is an integer, or a tuple of integers. which represents the shape of the new MLinExpr object.

Return value

new MLinExpr object.

Example

```
mexpr = MLinExpr.zeros((2,3))
x = model.addVar()
mexpr += x
```

MLinExpr.clear()

Synopsis

clear()

Description

Resets each element of the *MLinExpr Class* object to 0.0.

Example

```
mexpr = 2.0 * model.addMVar(3)
mexpr.clear()
```

MLinExpr.clone()

Synopsis

clone()

Description

Deep-copy a *MLinExpr Class* object.

Return value

new MLinExpr object

Example

```
mexpr = 2.0 * model.addMVar(3)
mexpr_copy = mexpr.clone()
```

MLinExpr.getValue()

Synopsis

```
getValue()
```

Description

Get the evaluation of each linear expression within the *MLinExpr Class* object.

Return value

Returns a NumPy ndarray of the same dimensions as the MLinExpr object whose elements are the evaluations of the corresponding expression.

Example

```
a = np.random.rand(4)
mx = m.addMVar((4, 3), nameprefix="mx")
mexpr = a @ mx
mc = m.addConstrs(mexpr <= 1.0)
model.solve()
print(mc.getValue())
```

MLinExpr.item()

Synopsis

```
item()
```

Description

Get the constraint object in 0-dimensional MLinExpr. Raises a ValueError exception if the MLinExpr object is not 0-dimensional.

Return value

Returns the linear constraint object.

Example

```
mexpr = 2.0 * model.addMVar(3)
print(mexpr[0].item())
```

MLinExpr.reshape()

Synopsis

```
reshape(shape, order='C')
```

Description

Returns a new MLinExpr object whose elements remain unchanged but whose shape is transformed by the argument shape.

Arguments

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MLinExpr object.

`order`

Optional parameter, the default is the character 'C', which means it is compatible with the C language, that is, it is stored in rows; it can also be set to the character 'F', that is, it is stored in columns, and it is compatible with the Fortune language.

Return value

Returns a new MLinExpr object with the same elements as the original MLinExpr object but with a different shape.

Example

```
mc = m.addConstrs(a @ mx <= b)
mc_2x2 = mc.reshape((2, 2))
```

MLinExpr.sum()

Synopsis

`sum(axis=None)`

Description

Sum the variables in the MLinExpr object, returning a new *MLinExpr Class* object.

Arguments

`axis`

Optional integer parameter, the default value is None, that is, to sum up variables one by one. Otherwise, sum over the given axis.

Return value

Returns an MLinExpr object representing the sum of the corresponding linear expressions.

Example

```
mexpr = 2.0 * model.addMVar((3, 5))
sum_all = mexpr.sum() #return 0-dimensional MLinExpr object
sum_row = mexpr.sum(axis = 0) #Return a 1-dimensional MLinExpr object with a shape␣
↪of (5, )
```

MLinExpr.tolist()

Synopsis

`tolist()`

Description

Converts an MLinExpr object to a one-dimensional list whose elements are linear expressions.

Return value

Return a 1D list containing *LinExpr Class*.

Example

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.tolist())
```

MLinExpr.transpose()

Synopsis

```
transpose()
```

Description

Generates a new MLinExpr object that is the transpose of the original MLinExpr object.

Return value

Returns the transposed MLinExpr object.

Example

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.transpose())
```

MLinExpr.ndim

Synopsis

```
ndim
```

Description

MLinExpr Class Dimensions of the object.

Return value

Integer value.

Example

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.ndim)
```

MLinExpr.shape

Synopsis

```
shape
```

Description

MLinExpr Class The shape of the object.

Return value

Integer tuple.

Example

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.shape)
```

MLinExpr.size

Synopsis

`size`

Description

The number of elements of *MLinExpr Class* object.

Return value

Integer value.

Example

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.size)
```

MLinExpr.T

Synopsis

`T`

Description

Transpose of *MLinExpr Class* object. Similar to the class method transpose().

Return value

Returns the transposed MLinExpr object.

Example

```
mexpr = 2.0 * model.addMVar((3, 5))
print(mexpr.T.shape) # The transposed shape is (5, 3)
```

MLinExpr.__eq__()

Synopsis

`__eq__()`

Description

Overload the == operator to build a *MConstrBuilder Class* object, which can be passed as the first argument to *Model.addConstrs*.

Return value

a *MConstrBuilder Class* object.

Example

```
model.addConstrs(A @ x == 1.0)
```

MLinExpr.__ge__()

Synopsis

```
__ge__()
```

Description

Overload the >= operator to build a *MConstrBuilder Class* object, which can be passed as the first argument to *Model.addConstrs*.

Return value

a *MConstrBuilder Class* object.

Example

```
model.addConstrs(A @ x >= 1.0)
```

MLinExpr.__le__()

Synopsis

```
__le__()
```

Description

Overload the <= operator to build a *MConstrBuilder Class* object, which can be passed as the first argument to *Model.addConstrs*.

Return value

a *MConstrBuilder Class* object.

Example

```
model.addConstrs(A @ x <= 1.0)
```

21.2.41 MQuadExpr Class

The MQuadExpr class is used in COPT to construct multi-dimensional quadratic expressions and supports NumPy's multi-dimensional array operations. The MQuadExpr object with an initial value of 0.0 can be generated by the class generation method `zeros()` , or by pairing two *MVar Class* objects are generated by (matrix) multiplication. The following member methods are provided:

MQuadExpr.zeros()

Synopsis

```
zeros(shape)
```

Description

This is the class generation method and can be called directly without an MQuadExpr object.

Arguments

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MQuadExpr object.

Return value

new MQuadExpr object.

Example

```
mqx = MQuadExpr.zeros((2,3)) # shape = (2, 3)
x = model.addVar()
mqx += 2.0 * x * x           # broadcast scalar
mqx += model.addMVar(3)      # broadcast MVar of shape (3,)
```

MQuadExpr.clear()

Synopsis

```
clear()
```

Description

Resets each element of the *MQuadExpr Class* object to 0.0.

Example

```
ma = model.addMVar(3, nameprefix='a')
mb = model.addMVar(3, nameprefix='b')
mqx = ma * mb    # elementwise multiply, shape = (3,)
mqx.clear()
print(mqx)       # result is [0.0, 0.0, 0.0]
```

MQuadExpr.clone()

Synopsis

```
clone()
```

Description

Deep-copy a *MQuadExpr Class* object.

Return value

new MQuadExpr object

Example

```
mx = model.addMVar((3, 3), nameprefix='mx')
mqx = 2.0 * mx @ mx      # matrix multiply, shape = (3, 3)
mqx_copy = mqx.clone()
mqx_copy.clear()
print(mqx)               # mqx is untouched
```

MQuadExpr.getValue()

Synopsis

```
getValue()
```

Description

Get the evaluation of each linear expression within the *MQuadExpr Class* object.

Return value

Returns a NumPy ndarray of the same dimensions as the MQuadExpr object whose elements are the evaluations of the corresponding expression.

Example

```
A = np.eye(3)
mx = m.addMVar(3, nameprefix="mx")
mqx = mx @ A @ mx         # 0-D MQuadExpr, shape = ()
m.addQConstr(mqx <= 9.0)
m.solve()
print(mqx.getValue())
```

MQuadExpr.item()

Synopsis

```
item()
```

Description

Get the constraint object in the 0-dimensional MQuadExpr. Raises a ValueError exception if the MQuadExpr object is not 0-dimensional.

Return value

Returns the linear constraint object.

Example

```
x = m.addVar()
mqx = MQuadExpr.zeros(3) + x * x
print(mqx[1].item()) # Return QuadExpr(x * x)
```

MQuadExpr.reshape()

Synopsis

```
reshape(shape, order='C')
```

Description

Returns a new MQuadExpr object whose elements are unchanged but whose shape is transformed by the argument shape.

Arguments

`shape`

The value is an integer, or a tuple of integers. which represents the shape of the new MQuadExpr object.

`order`

Optional parameter, the default is the character 'C', which means it is compatible with the C language, that is, it is stored in rows; it can also be set to the character 'F', that is, it is stored in columns, and it is compatible with the Fortune language.

Return value

Returns a new MQuadExpr object with the same elements as the original MQuadExpr object but with a different shape.

Example

```
mqx = MQuadExpr.zeros(6)
mqx_2x3 = mqx.reshape((2, 3))
```

MQuadExpr.sum()

Synopsis

```
sum(axis=None)
```

Description

Sums the variables in the MQuadExpr object, returning a new *MQuadExpr Class* object.

Arguments

`axis`

Optional integer parameter, the default value is None, that is, to sum up variables one by one. Otherwise, sum over the given axis.

Return value

Returns an MQuadExpr object representing the sum of the corresponding linear expressions.

Example

```
ma = model.addMVar((2, 3), nameprefix='ma')
mb = model.addMVar((3, 2), nameprefix='mb')
mqx = ma @ mb
sum_all = mqx.sum() # Return 0-dimensional MQuadExpr object
sum_row = mqx.sum(axis = 0) # Return a 1-dimensional MQuadExpr object with a shape␣
↪of (2, )
```

MQuadExpr.tolist()

Synopsis

```
tolist()
```

Description

Converts an MQuadExpr object to a one-dimensional list whose elements are linear expressions.

Return value

Return a 1D list containing *LinExpr Class*.

Example

```
print(MQuadExpr.zeros((2,3)).tolist()) # a list of length 6
```

MQuadExpr.transpose()

Synopsis

```
transpose()
```

Description

Generates a new MQuadExpr object that is the transpose of the original MQuadExpr object.

Return value

Returns the transposed MQuadExpr object.

Example

```
mqx = MQuadExpr.zeros((2,3))
print(mqx.transpose().shape) # shape = (3, 2)
```

MQuadExpr.ndim

Synopsis

ndim

Description

MQuadExpr Class Dimensions of the object.

Return value

Integer value.

Example

```
mqx = MQuadExpr.zeros((2,3))
print(mqx.ndim)  # ndim = 2
```

MQuadExpr.shape

Synopsis

shape

Description

MQuadExpr Class The shape of the object.

Return value

Integer tuple.

Example

```
print(MQuadExpr.zeros((2,3)).shape) # shape = (2, 3)
```

MQuadExpr.size

Synopsis

size

Description

The number of elements of *MQuadExpr Class* object.

Return value

Integer value.

Example

```
mqx = MQuadExpr.zeros((2,3))
print(mqx.size)  # size= 6
```

MQuadExpr.T

Synopsis

```
T
```

Description

Transpose of *MQuadExpr Class* object. Similar to the class method transpose().

Return value

Returns the transposed MQuadExpr object.

Example

```
mqx = MQuadExpr.zeros((2,3))
print(mqx.T.shape) # shape = (3, 2)
```

MQuadExpr.__eq__()

Synopsis

```
__eq__()
```

Description

Overload the == operator to build a *MQConstrBuilder Class* object, which can be passed as the first argument to *Model.addQConstr*.

Return value

a *MQConstrBuilder Class* object.

Example

```
model.addQConstr(x @ Q @ y == 1.0)
```

MQuadExpr.__ge__()

Synopsis

```
__ge__()
```

Description

Overload the >= operator to build a *MQConstrBuilder Class* object, which can be passed as the first argument to *Model.addQConstr*.

Return value

a *MQConstrBuilder Class* object.

Example

```
model.addQConstr(x @ Q @ y >= 1.0)
```

MQuadExpr.__le__()

Synopsis

```
__le__()
```

Description

Overload the <= operator to build a *MQConstrBuilder Class* object, which can be passed as the first argument to *Model.addQConstr*.

Return value

a *MQConstrBuilder Class* object.

Example

```
model.addQConstr(x @ Q @ y <= 1.0)
```

21.2.42 ExprBuilder Class

ExprBuilder object contains operations related to building linear expressions, and provides the following methods:

ExprBuilder()

Synopsis

```
ExprBuilder(arg1=0.0, arg2=None)
```

Description

Create a *ExprBuilder Class* object.

If argument `arg1` is constant, argument `arg2` is `None`, then create a *ExprBuilder Class* object and initialize it using argument `arg1`. If argument `arg1` is *Var Class* or *ExprBuilder Class* object, and argument `arg2` is constant or considered to be constant 1.0 when argument `arg2` is `None`, then initialize the newly created *ExprBuilder Class* object using arguments `arg1` and `arg2`. If argument `arg1` and `arg2` are list objects, then they are variables and coefficients used to initialize the newly created *ExprBuilder Class* object.

Arguments

`arg1`

Optional, 0.0 by default.

`arg2`

Optional, `None` by default.

Example

```
# Create a new ExprBuilder object and initialize it to 0.0
expr0 = ExprBuilder()
# Create a ExprBuilder object and initialize it to x + 2*y
expr2 = ExprBuilder([x, y], [1, 2])
```

ExprBuilder.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of terms in an expression builder.

Example

```
# Retrieve the number of terms in expression builder 'expr'
exprsize = expr.getSize()
```

ExprBuilder.getCoeff()

Synopsis

```
getCoeff(idx)
```

Description

Retrieve the coefficient of a variable by its index from an expression builder.

Arguments

`idx`

Index of the variable in the expression builder, starting with 0.

Example

```
# Retrieve the coefficient for the term at index 1 from expression builder 'expr'
coeff = expr.getCoeff(1)
```

ExprBuilder.getVar()

Synopsis

```
getVar(idx)
```

Description

Retrieve the variable by its index from an expression builder. Return a *Var Class* object.

Arguments

`idx`

Index of the variable in the expression builder, starting with 0.

Example

```
# Retrieve the variable for the term at index 1 from expression builder 'expr'
x = expr.getVar(1)
```

ExprBuilder.getConstant()

Synopsis

```
getConstant()
```

Description

Retrieve the constant term from an expression builder.

Example

```
# Retrieve the constant term from linear expression builder 'expr'
constant = expr.getConstant()
```

ExprBuilder.addTerm()

Synopsis

```
addTerm(var, coeff=1.0)
```

Description

Add a new term to current expression builder.

Arguments

`var`

Variable to add.

`coeff`

Magnification coefficient for added term. Optional, 1.0 by default.

Example

```
# Add term 2*x to linear expression builder 'expr'
expr.addTerm(x, 2.0)
```

ExprBuilder.addExpr()

Synopsis

```
addExpr(expr, coeff=1.0)
```

Description

Add new expression builder to the current one.

Arguments

`expr`

Expression builder to add.

`coeff`

Magnification coefficients for the added expression builder. Optional, 1.0 by default.

Example

```
# Add linear expression builder 2*x + 2*y to 'expr'
expr.addExpr(x + y, 2.0)
```

ExprBuilder.clone()

Synopsis

```
clone()
```

Description

Create a deep copy of the expression builder.

Example

```
# Create a deep copy of expression builder 'expr'
exprcopy = expr.clone()
```

ExprBuilder.getExpr()

Synopsis

```
getExpr()
```

Description

Create a linear expression related to the expression builder. Returns a *LinExpr Class* object.

Example

```
# Get the linear expression object related to expression builder 'exprbuilder'
expr = exprbuilder.getExpr()
```

21.2.43 LinExpr Class

LinExpr object contains operations related to variables for building linear constraints, and provides the following methods:

LinExpr()

Synopsis

```
LinExpr(arg1=0.0, arg2=None)
```

Description

Create a *LinExpr Class* object.

If argument `arg1` is constant, argument `arg2` is `None`, then create a *LinExpr Class* object and initialize it using argument `arg1`. If argument `arg1` is *Var Class* or *LinExpr Class* object, and argument `arg2` is constant or considered to be constant 1.0 when argument `arg2` is `None`, then initialize the newly created *LinExpr Class* object using arguments `arg1` and `arg2`. If argument `arg1` is list object and argument `arg2` is `None`, then argument `arg1` contains a list of variable-coefficient pairs and initialize the newly created *LinExpr Class* object using arguments `arg1` and `arg2`. For other forms of arguments, call method `addTerms` to initialize the newly created *LinExpr Class* object.

Arguments

`arg1`

Optional, 0.0 by default.

`arg2`

Optional, `None` by default.

Example

```
# Create a new LinExpr object and initialize it to 0.0
expr0 = LinExpr()
# Create a LinExpr object and initialize it to 2*x + 3*y
expr1 = LinExpr([(x, 2), (y, 3)])
# Create a LinExpr object and initialize it to x + 2*y
expr2 = LinExpr([x, y], [1, 2])
```

LinExpr.setCoeff()

Synopsis

`setCoeff(idx, newval)`

Description

Set new coefficient value of a variable based on its index in an expression.

Arguments

`idx`

Index of the variable in the expression, starting with 0.

`newval`

New coefficient value of the variable.

Example

```
# Set the coefficient for the term at index 0 in expression expr to 1.0
expr.setCoeff(0, 1.0)
```

LinExpr.getCoeff()

Synopsis

`getCoeff(idx)`

Description

Retrieve the coefficient of a variable by its index from an expression.

Arguments

`idx`

Index of the variable in the expression, starting with 0.

Example

```
# Retrieve the coefficient for the term at index 1 from expression expr
coeff = expr.getCoeff(1)
```

LinExpr.getVar()

Synopsis

```
getVar(idx)
```

Description

Retrieve the variable by its index from an expression. Return a *Var Class* object.

Arguments

`idx`

Index of the variable in the expression, starting with 0.

Example

```
# Retrieve the variable for the term at index 1 from expression expr
x = expr.getVar(1)
```

LinExpr.getConstant()

Synopsis

```
getConstant()
```

Description

Retrieve the constant term from an expression.

Example

```
# Retrieve the constant term from linear expression expr
constant = expr.getConstant()
```

LinExpr.getValue()

Synopsis

```
getValue()
```

Description

Retrieve the value of an expression computed using the current solution.

Example

```
# Retrieve the value of expression expr for the current solution.
val = expr.getValue()
```

LinExpr.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of terms in an expression.

Example

```
# Retrieve the number of terms in expression expr
exprsize = expr.getSize()
```

LinExpr.setConstant()

Synopsis

```
setConstant(newval)
```

Description

Set the constant term of linear expression.

Arguments

`newval`

Constant term to be set.

Example

```
# Set constant term of linear expression 'expr' to 2.0
expr.setConstant(2.0)
```

LinExpr.addConstant()

Synopsis

```
addConstant(newval)
```

Description

Add a constant to an expression.

Arguments

`newval`

Constant to add.

Example

```
# Add constant 2.0 to linear expression 'expr'
expr.addConstant(2.0)
```

LinExpr.addTerm()

Synopsis

```
addTerm(var, coeff=1.0)
```

Description

Add a new term to current expression.

Arguments

`var`

Variable to add.

`coeff`

Magnification coefficient for added term. Optional, 1.0 by default.

Example

```
# Add term x to linear expression 'expr'
expr.addTerm(x)
```

LinExpr.addTerms()

Synopsis

```
addTerms(vars, coeffs)
```

Description

Add a single term or multiple terms into an expression.

If argument `vars` is *Var Class* object, then argument `coeffs` is constant; If argument `vars` is *VarArray Class* object or list, then argument `coeffs` is constant or list; If argument `vars` is dictionary or *tupledict Class* object, then argument `coeffs` is constant, dict, or *tupledict Class* object.

Arguments

`vars`

Variables to add.

`coeffs`

Coefficients for variables.

Example

```
# Add term 2*x + 2*y to linear expression 'expr'
expr.addTerms([x, y], [2.0, 3.0])
```

LinExpr.addExpr()

Synopsis

```
addExpr(expr, coeff=1.0)
```

Description

Add new expression to the current one.

Arguments

`expr`

Expression or expression builder to add.

`coeff`

Magnification coefficients for the added expression. Optional, 1.0 by default.

Example

```
# Add linear expression 2*x + 2*y to 'expr'
expr.addExpr(x + y, 2.0)
```

LinExpr.clone()

Synopsis

```
clone()
```

Description

Create a deep copy of the expression.

Example

```
# Create a deep copy of expression expr
exprcopy = expr.clone()
```

LinExpr.reserve()

Synopsis

```
reserve(n)
```

Description

Pre-allocate space for linear expression object.

Arguments

`n`

Number of terms to be allocated.

Example

```
# Allocate 100 terms for linear expression 'expr'
expr.reserve(100)
```

LinExpr.remove()

Synopsis

```
remove(item)
```

Description

Remove a term from a linear expression.

If argument `item` is constant, then remove the term stored at index i of the expression; otherwise argument `item` is *Var Class* object.

Arguments

`item`

Constant index or variable of the term to be removed.

Example

```
# Remove the term whose index is 2 from linear expression expr
expr.remove(2)
# Remove the term whose variable is x from linear expression expr
expr.remove(x)
```

21.2.44 QuadExpr Class

QuadExpr object contains operations related to variables for building linear constraints, and provides the following methods:

QuadExpr()

Synopsis

`QuadExpr(expr=0.0)`

Description

Create a *QuadExpr Class* object.

Argument `expr` is constant, *Var Class*, *LinExpr Class* object or *QuadExpr Class* object.

Arguments

`expr`

Optional, 0.0 by default.

Example

```
# Create a new QuadExpr object and initialize it to 0.0
quadexpr0 = QuadExpr()
# Create a QuadExpr object and initialize it to 2*x + 3*y
quadexpr1 = QuadExpr([(x, 2), (y, 3)])
# Create a QuadExpr object and initialize it to x*x + 2*x*y
quadexpr2 = QuadExpr(x*x + 2*x*y)
```

QuadExpr.setCoeff()

Synopsis

`setCoeff(idx, newval)`

Description

Set new coefficient value of a term based on its index in a quadratic expression.

Arguments

`idx`

Index of the term in the quadratic expression, starting with 0.

`newval`

New coefficient value of the term.

Example

```
# Set the coefficient for the term at index 0 in quadratic expression quadexpr to 1.0
quadexpr.setCoeff(0, 1.0)
```

QuadExpr.getCoeff()

Synopsis

```
getCoeff(idx)
```

Description

Retrieve the coefficient of a term by its index from a quadratic expression.

Arguments

`idx`

Index of the term in the quadratic expression, starting with 0.

Example

```
# Retrieve the coefficient for the term at index 1 from quadratic expression quadexpr
coeff = quadexpr.getCoeff(1)
```

QuadExpr.getVar1()

Synopsis

```
getVar1(idx)
```

Description

Retrieve the first variable of a quadratic term by its index from an expression. Return a *Var Class* object.

Arguments

`idx`

Index of the quadratic term in the expression, starting with 0.

Example

```
# Retrieve the first variable of a quadratic term at index 1 from quadratic expression quadexpr
x = quadexpr.getVar1(1)
```

QuadExpr.getVar2()

Synopsis

```
getVar2(idx)
```

Description

Retrieve the second variable of a quadratic term by its index from an expression. Return a *Var Class* object.

Arguments

`idx`

Index of the quadratic term in the expression, starting with 0.

Example

```
# Retrieve the first variable of a quadratic term at index 1 from quadratic expression quadexpr
y = quadexpr.getVar2(1)
```

QuadExpr.getLinExpr()

Synopsis

```
getLinExpr()
```

Description

Retrieve the linear terms (if exist) fronm quadratic expression. Return a *LinExpr Class* object.

Example

```
# Retrieve the linear terms from a quadratic expression quadexpr
linexpr = quadexpr.getLinExpr()
```

QuadExpr.getConstant()

Synopsis

```
getConstant()
```

Description

Retrieve the constant term from a quadratic expression.

Example

```
# Retrieve the constant term from quadratic expression quadexpr
constant = quadexpr.getConstant()
```

QuadExpr.getValue()

Synopsis

```
getValue()
```

Description

Retrieve the value of a quadratic expression computed using the current solution.

Example

```
# Retrieve the value of quadratic expression quadexpr for the current solution.
val = quadexpr.getValue()
```

QuadExpr.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of terms in a quadratic expression.

Example

```
# Retrieve the number of terms in quadratic expression quadexpr
exprsize = quadexpr.getSize()
```

QuadExpr.setConstant()

Synopsis

`setConstant(newval)`

Description

Set the constant term of quadratic expression.

Arguments

`newval`

Constant to set.

Example

```
# Set constant term of quadratic expression 'quadexpr' to 2.0
quadexpr.setConstant(2.0)
```

QuadExpr.addConstant()

Synopsis

`addConstant(newval)`

Description

Add a constant to a quadratic expression.

Arguments

`newval`

Constant to add.

Example

```
# Add constant 2.0 to quadratic expression 'quadexpr'
quadexpr.addConstant(2.0)
```

QuadExpr.addTerm()

Synopsis

`addTerm(coeff, var1, var2=None)`

Description

Add a new term to current quadratic expression.

Arguments

`coeff`

Magnification coefficient for added term. Optional, 1.0 by default.

`var1`

The first variable for added term.

`var2`

The second variable for added term, defaults to `None`, i.e. add a linear term.

Example

```
# Add term x to quadratic expression 'quadexpr'
quadexpr.addTerm(1.0, x)
```

QuadExpr.addTerms()

Synopsis

```
addTerms(coeffs, vars1, vars2=None)
```

Description

Add a single term or multiple terms into a quadratic expression.

If argument `vars` is *Var Class* object, then argument `vars2` is *Var Class* object or `None`, argument `coeffs` is constant; If argument `vars` is *VarArray Class* object or list, then argument `vars2` is *VarArray Class* object, list or `None`, argument `coeffs` is constant or list; If argument `vars` is dictionary or *tupledict Class* object, then argument `vars2` is dictionary, *tupledict Class* object or `None`, argument `coeffs` is constant, dictionary, or *tupledict Class* object.

Arguments

`coeffs`

Coefficients for terms.

`vars1`

The first variable of each term.

`vars2`

The second variable of each term, defaults to `None`, i.e. add a linear term.

Example

```
# Add term 2*x + 3y + 2*x*x + 3*x*y to quadratic expression 'quadexpr'
# Note: Mixed format is supported by addTerms yet.
quadexpr.addTerms([2.0, 3.0], [x, y])
quadexpr.addTerms([2.0, 3.0], [x, x], [x, y])
```

QuadExpr.addLinExpr()

Synopsis

```
addLinExpr(expr, mult=1.0)
```

Description

Add new linear expression to the current quadratic expression.

Arguments

`expr`

Linear expression or linear expression builder to add.

`mult`

Magnification coefficient for the added expression. Optional, 1.0 by default.

Example

```
# Add linear expression 2*x + 2*y to 'quadexpr'
quadexpr.addLinExpr(x + y, 2.0)
```

QuadExpr.addQuadExpr()

Synopsis

```
addQuadExpr(expr, mult=1.0)
```

Description

Add new quadratic expression to the current one.

Arguments

`expr`

Expression or expression builder to add.

`mult`

Magnification coefficients for the added expression. Optional, 1.0 by default.

Example

```
# Add quadratic expression x*x + 2*y to 'quadexpr'
quadexpr.addQuadExpr(x*x + 2*y, 2.0)
```

QuadExpr.clone()

Synopsis

```
clone()
```

Description

Create a deep copy of the expression.

Example

```
# Create a deep copy of quadratic expression quadexpr
exprcopy = quadexpr.clone()
```

QuadExpr.reserve()

Synopsis

```
reserve(n)
```

Description

Pre-allocate space for quadratic expression object.

Arguments

`n`

Number of terms to be allocated.

Example

```
# Allocate 100 terms for quadratic expression 'expr'
expr.reserve(100)
```

QuadExpr.remove()

Synopsis

```
remove(item)
```

Description

Remove a term from a quadratic expression.

If argument `item` is constant, then remove the term stored at index i of the expression; otherwise argument `item` is *Var Class* object.

Arguments

`item`

Constant index or variable of the term to be removed.

Example

```
# Remove the term whose index is 2 from quadratic expression quadexpr
quadexpr.remove(2)
# Remove the terms one of which variable is x from quadratic expression quadexpr
quadexpr.remove(x)
```

21.2.45 PsdExpr Class

PsdExpr object contains operations related to variables for building positive semi-definite constraints, and provides the following methods:

PsdExpr()

Synopsis

```
PsdExpr(expr=0.0)
```

Description

Create a *PsdExpr Class* object.

Arguments

`expr`

Optional, 0.0 by default, which can be a constant, *Var Class*, *LinExpr Class* object or *PsdExpr Class* object.

Example

```
# Create a new PsdExpr object and initialize it to 0.0
expr0 = PsdExpr()
# Create a PsdExpr object and initialize it to 2*x + 3*y
expr1 = PsdExpr(2*x + 3*y)
```

PsdExpr.setCoeff()

Synopsis

```
setCoeff(idx, mat)
```

Description

Set new symmetric matrix coefficient for a positive semi-definite variable based on its index in the expression.

Arguments

`idx`

Index of the positive semi-definite variable in the expression, starting with 0.

`mat`

New symmetric matrix coefficient of the positive semi-definite variable.

Example

```
# Set symmetric matrix for the positive semi-definite variable at index 0 in expression "expr"␣
↪to mat
expr.setCoeff(0, mat)
```

PsdExpr.getCoeff()

Synopsis

```
getCoeff(idx)
```

Description

Retrieve the symmetric matrix coefficient of a positive semi-definite variable by its index from the expression.

Arguments

`idx`

Index of the positive semi-definite variable in the expression, starting with 0.

Example

```
# Retrieve the symmetric matrix coefficient for the positive semi-definite variable at index 1␣
↪from expression expr
mat = expr.getCoeff(1)
```

PsdExpr.getPsdVar()

Synopsis

```
getPsdVar(idx)
```

Description

Retrieve a positive semi-definite variable by its index from the expression. Return a *PsdVar Class* object.

Arguments

`idx`

Index of the positive semi-definite variable in the expression, starting with 0.

Example

```
# Retrieve the positive semi-definite variable at index 1 from expression expr
x = expr.getPsdVar(1)
```

PsdExpr.getLinExpr()

Synopsis

```
getLinExpr()
```

Description

Retrieve the linear terms (if exist) fronm positive semi-definite expression. Return a *LinExpr Class* object.

Example

```
# Retrieve the linear terms from a positive semi-definite expression expr
linexpr = expr.getLinExpr()
```

PsdExpr.getConstant()

Synopsis

```
getConstant()
```

Description

Retrieve the constant term from a positive semi-definite expression.

Example

```
# Retrieve the constant term from expression expr
constant = expr.getConstant()
```

PsdExpr.getValue()

Synopsis

```
getValue()
```

Description

Retrieve the value of a positive semi-definite expression computed using the current solution.

Example

```
# Retrieve the value of positive semi-definite expression expr for the current solution.
val = expr.getValue()
```

PsdExpr.getSize()

Synopsis

```
getSize()
```

Description

Retrieve the number of terms in a positive semi-definite expression.

Example

```
# Retrieve the number of terms in expression expr
exprsize = expr.getSize()
```

PsdExpr.setConstant()

Synopsis

```
setConstant(newval)
```

Description

Set the constant term of positive semi-definite expression.

Arguments

`newval`

Constant to set.

Example

```
# Set constant term of expression 'expr' to 2.0
expr.setConstant(2.0)
```

PsdExpr.addConstant()

Synopsis

```
addConstant(newval)
```

Description

Add a constant to a positive semi-definite expression.

Arguments

`newval`

Constant to add.

Example

```
# Add constant 2.0 to expression 'expr'
expr.addConstant(2.0)
```

PsdExpr.addTerm()

Synopsis

```
addTerm(var, mat)
```

Description

Add a new term to current positive semi-definite expression.

Arguments

`var`

The positive semi-definite variable to add.

`mat`

The symmetric matrix coefficient for the positive semi-definite variable.

Example

```
# Add positive semi-definite term C1 * X to expression 'expr'
expr.addTerm(X, C1)
```

PsdExpr.addTerms()

Synopsis

```
addTerms(vars, mats)
```

Description

Add a single term or multiple positive semi-definite terms into a positive semi-definite expression.

If argument `vars` is *PsdVar Class* object, then argument `mats` is *SymMatrix Class* object; If argument `vars` is *PsdVarArray Class* object or list, then argument `mats` is *SymMatrixArray Class* or list;

Arguments

`vars`

The positive semi-definite variables to add.

`mats`

The symmetric matrices of the positive semi-definite terms.

Example

```
# Add terms C1 * X1 + C2 * X2 to expression 'expr'
expr.addTerms([X1, X2], [C1, C2])
```

PsdExpr.addLinExpr()

Synopsis

```
addLinExpr(expr, mult=1.0)
```

Description

Add new linear expression to the current positive semi-definite expression.

Arguments

`expr`

Linear expression or linear expression builder to add.

`mult`

Magnification coefficient for the added expression. Optional, 1.0 by default.

Example

```
# Add linear expression 2*x + 2*y to 'expr'
expr.addLinExpr(x + y, 2.0)
```

PsdExpr.addPsdExpr()

Synopsis

```
addPsdExpr(expr, mult=1.0)
```

Description

Add new positive semi-definite expression to the current one.

Arguments

`expr`

Positive semi-definite expression or positive semi-definite expression builder to add.

`mult`

Magnification coefficient for the added positive semi-definite expression. Optional, 1.0 by default.

Example

```
# Add positive semi-definite expression C * X to 'expr'
expr.addPsdExpr(C*X)
```

PsdExpr.clone()

Synopsis

```
clone()
```

Description

Create a deep copy of the expression.

Example

```
# Create a deep copy of expression expr
exprcopy = expr.clone()
```

PsdExpr.reserve()

Synopsis

```
reserve(n)
```

Description

Pre-allocate space for positive semi-definite expression object.

Arguments

`n`

Number of terms to be allocated.

Example

```
# Allocate 100 terms for positive semi-definite expression 'expr'
expr.reserve(100)
```

PsdExpr.remove()

Synopsis

```
remove(item)
```

Description

Remove a term from a positive semi-definite expression.

If argument `item` is constant, then remove the term stored at index i of the expression; otherwise argument `item` is *PsdVar Class* object.

Arguments

`item`

Constant index or *PsdVar Class* variable of the term to be removed.

Example

```
# Remove the term whose index is 2 from positive semi-definite expression expr
expr.remove(2)
# Remove the terms one of which variable is x from positive semi-definite expression expr
expr.remove(x)
```

21.2.46 CallbackBase Class

COPT CallbackBase class. This is an abstract class, the user needs to implement the function *CallbackBase.callback()* to create an instance. The instance is passed in as the first argument of the method *Model.setCallback()*

CallbackBase.where()

Synopsis

```
where()
```

Description

Get context in callback.

Return value

Returns an integer value.

CallbackBase.callback()

Synopsis

```
callback()
```

Description

The callback function is a pure virtual function which needs to be implemented by the user. The user can describe the information that needs to be obtained or the operation that needs to be performed during the solution process.

Example

```
class CoptCallback(CallbackBase):
    def __init__(self):
        super().__init__()
    def callback(self):
        # Get the objective value when finding a feasible MIP solution
        if self.where() == COPT.CBCONTEXT_MIPSOL:
            db = self.getInfo(COPT.CBInfo.MipCandObj)
```

CallbackBase.interrupt()

Synopsis

```
interrupt()
```

Description

Interrupt the callback process.

CallbackBase.addUserCut()

Synopsis

```
addUserCut(lhs, sense = None, rhs = None)
```

Description

Add a user cut to the MIP model from within the callback function.

Arguments

`lhs`

Left-hand side expression for the new user cut. It can take the value of *Var Class* object, *LinExpr Class* object, or *ConstrBuilder Class* .

`sense`

The sense of the new user cut. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

Optional. None by default.

The user cut added from within callback can only have a single comparison operator.

`rhs`

Right-hand side expression for the new user cut.

Optional. None by default.

It can be a constant, or *Var Class* object, or *LinExpr Class* object.

Example

```
self.addUserCut(x+y <= 1)
```

CallbackBase.addUserCuts()

Synopsis

```
addUserCuts(generator)
```

Description

Add a set of user cuts to the MIP model from within the callback function.

Arguments

`generator`

Array of builders for user cuts. It can be *ConstrBuilderArray Class* object.

Example

```
self.addUserCuts(x[i]+y[i] <= 1 for i in range(10))
```

CallbackBase.addLazyConstr()

Synopsis

```
addLazyConstr(lhs, sense = None, rhs = None)
```

Description

Add a lazy constraint to the MIP model from within the callback function.

Arguments

`lhs`

Left-hand side expression for the new lazy constraint. It can take the value of *Var Class* object, *LinExpr Class* object or *ConstrBuilder Class* object.

`sense`

The sense of the lazy constraint. It supports for `LESS_EQUAL`, `GREATER_EQUAL`, `EQUAL` and `FREE` .

Optional. None by default.

The lazy constraint added from within callback can only have a single comparison operator.

`rhs`

Right-hand side expression for the new lazy constraint.

Optional. None by default.

It can be a constant, or *Var Class* object, or *LinExpr Class* object.

Example

```
self.addLazyConstr(x+y <= 1)
```

CallbackBase.addLazyConstrs()

Synopsis

`addLazyConstrs(generator)`

Description

Add a set of lazy constraints to the MIP model from within the callback function.

Arguments

`generator`

Array of builders for lazy constraints. It can be *ConstrBuilderArray Class* object.

Example

```
self.addLazyConstrs(x[i] + y[i] <= 1 for i in range(10))
```

CallbackBase.getInfo()

Synopsis

`getInfo(cbinfo)`

Description

Retrieve the value of the specified callback information.

Arguments

`cbinfo`

The name of the callback information. Please refer to *Callback Information* for possible values.

Return value

Returns a constant(int-valued or double-valued).

Example

```
db = self.getInfo(COPT.CBInfo.BestBnd)
```

CallbackBase.getRelaxSol()

Synopsis

```
getRelaxSol(vars)
```

Description

Retrieve the LP-relaxation solution of the specified variables at the current node.

Note that this method can only be invoked if `CallbackBase.where() == COPT.CBCONTEXT_MIPRELAX`.

Arguments

`vars`

The variables to retrieve the LP-relaxation solution values.

Return value

When parameter `vars` is *Var Class* object, it returns a constant, which is the LP-relaxation solution value of the specified variable.

When parameter `vars` is list or *VarArray Class* object, it returns a list of constants, consisting of the solution of the specified variables.

When parameter `args` is dictionary or *tupledict Class* object, it returns *tupledict Class* object(the indices of specified variables as key, the solutions of the specified variables as value).

When parameter `args` is `None` , it returns the LP-relaxation solution values of all variables.

Example

```
vals = self.getRelaxSol(vars)
```

CallbackBase.getIncumbent()

Synopsis

```
getIncumbent(vars)
```

Description

Retrieve values from the best feasible solution of the specified variables.

Arguments

`vars`

The variables whose solution values to retrieve.

Return value

When parameter `vars` is *Var Class* object, it returns a constant, which is the solution value of the specified variable.

When parameter `vars` is list or *VarArray Class* object, it returns a list of constants, consisting of the solution of the specified variables.

When parameter `args` is dictionary or *tupledict Class* object, it returns *tupledict Class* object(the indices of specified variables as key, the solutions of the specified variables as value).

When parameter `args` is `None` , it returns the incumbent values of all variables.

Example

```
vals = self.getIncumbent(vars)
```

CallbackBase.getSolution()

Synopsis

```
getSolution(vars)
```

Description

Retrieve values from the current solution of the specified variables.

Note that this method can only be invoked if `CallbackBase.where() == COPT.CBCONTEXT_MIPSOL`.

Arguments

`vars`

The variables whose solution values to retrieve.

Return value

When parameter `vars` is *Var Class* object, it returns a constant, which is the solution value of the specified variable.

When parameter `vars` is list or *VarArray Class* object, it returns a list of constants, consisting of the solution of the specified variables.

When parameter `args` is dictionary or *tupledict Class* object, it returns *tupledict Class* object(the indices of specified variables as key, the solutions of the specified variables as value).

When parameter `args` is `None` , it returns the solution values of all variables.

Example

```
vals = self.getSolution(vars)
```

CallbackBase.setSolution()

Synopsis

```
setSolution(vars, vals)
```

Description

Set heuristic solution values for the specified variables.

When parameter `vars` is *Var Class* object, parameter `vals` is constant;

When parameter `vars` is dictionary or *tupledict Class* object, parameter `vals` can be constant, dictionary or *tupledict Class* object;

When parameter `vars` is list, *VarArray Class* object, parameter `vals` can be constant or list.

Arguments

`vars`

The variables to be set value.

`vals`

The values of the variables in the solution.

Example

```
self.setSolution(x, 1)
```

CallbackBase.loadSolution()

Synopsis

```
loadSolution()
```

Description

Load the currently heuristic solution into model.

Note that a complete solution is required here.

Example

```
self.loadSolution()
```

21.2.47 GenConstrX Class

In the `Model` class, the constraints added by `addGenConstrXXX` (such as: `addGenConstrMax`) will return a `GenConstrX` object.

GenConstrX.getAttr()

Synopsis

```
getAttr(attrname)
```

Description

Get the attribute value of the `GenConstrX` class object, support to get the type and name of the `GenConstrX` class object.

Example

```
# Get the name of con_max
con_max.getAttr("name")
# Get the type of con_max
con_max.getAttr("type")
```

GenConstrX.setAttr()

Synopsis

```
setAttr(attrname)
```

Arguments

Set the attribute value of the `GenConstrX` class object, and support setting the name of the `GenConstrX` class object.

Example

```
# Set the name of con_max
con_max.setAttr("name")
```

21.2.48 CoptError Class

CoptError Class provides operations on error. An exception of the CoptError is thrown when error occurs in a method call corresponding to the underlying interface of solver. The following attributes are provided to retrieve error information:

- CoptError.retcode

  Error code.

- CoptError.message

  Error message.

21.3 Helper Functions and Utilities

Helper functions and utilities are encapsulated based on Python's basic data types, providing easy-to-use data types to facilitate the rapid construction of complex optimization models. This section will explain its functions and usages.

21.3.1 Helper Functions

multidict()

Synopsis

`multidict(data)`

Description

Split a single dictionary into keys and multiple dictionaries. Return keys and dictionaries.

Arguments

`data`

A Python dictionary to be applied. Each key should map to a list of n values.

Example

```
keys, dict1, dict2 = multidict({
  "hello": [0, 1],
  "world": [2, 3]})
```

quicksum()

Synopsis

`quicksum(data)`

Description

Build expressions efficiently. Return a *LinExpr Class* object.

Arguments

`data`

Terms to add.

Example

```
expr = quicksum(m.getVars())
```

21.3.2 tuplelist Class

The tuplelist object is an encapsulation based on Python lists, and provides the following methods:

tuplelist()

Synopsis

`tuplelist(list)`

Description

Create and return a *tuplelist Class* object.

Arguments

`list`

A Python list.

Example

```
tl = tuplelist([(0, 1), (1, 2)])
tl = tuplelist([('a', 'b'), ('b', 'c')])
```

tuplelist.add()

Synopsis

`add(item)`

Description

Add an item to a *tuplelist Class* object

Arguments

`item`

Item to add, which can be a Python tuple.

Example

```
tl = tuplelist([(0, 1), (1, 2)])
tl.add((2, 3))
```

tuplelist.select()

Synopsis

`select(pattern)`

Description

Get all terms that match the specified pattern. Return a *tuplelist Class* object.

Arguments

`pattern`

Specified pattern.

Example

```
tl = tuplelist([(0, 1), (0, 2), (1, 2)])
tl.select(0, '*')
```

21.3.3 tupledict Class

The tupledict class is an encapsulation based on Python dictionaries, and provides the following methods:

tupledict()

Synopsis

`tupledict(args, kwargs)`

Description

Create and return a *tupledict Class* object.

Arguments

`args`

Positional arguments.

`kwargs`

Named arguments.

Example

```
d = tupledict([(0, "hello"), (1, "world")])
```

tupledict.select()

Synopsis

`select(pattern)`

Description

Get all terms that match the specified pattern. Return a *tupledict Class* object.

Arguments

`pattern`

Specified pattern.

Example

```
d = tupledict([(0, "hello"), (1, "world")])
d.select()
```

tupledict.sum()

Synopsis

```
sum(pattern)
```

Description

Sum all terms that match the specified pattern. Return a *LinExpr Class* object.

Arguments

`pattern`

Specified pattern.

Example

```
expr = x.sum()
```

tupledict.prod()

Synopsis

```
prod(coeff, pattern)
```

Description

Filter terms that match the specified pattern and multiply by coefficients. Return a *LinExpr Class* object.

Arguments

`coeff`

Coefficients, which can be a dict or a *tupledict Class* object.

`pattern`

Specified pattern.

Example

```
coeff = dict([(1, 0.1), (2, 0.2)])
expr  = x.prod(coeff)
```

21.3.4 ProbBuffer Class

The ProbBuffer is an encapsulation of buffer of string stream, and provides the following methods:

ProbBuffer()

Synopsis

```
ProbBuffer(buff)
```

Description

Create and return a *ProbBuffer Class* object.

Arguments

`buff`

Size of buffer, defaults to `None`, i.e. the buffer size is 0.

Example

```
# Create a buffer of size 100
buff = ProbBuffer(100)
```

ProbBuffer.getData()

Synopsis

```
getData()
```

Description

Get the contents of buffer.

Example

```
# Print the contents in buffer
print(buff.getData())
```

ProbBuffer.getSize()

Synopsis

```
getSize()
```

Description

Get the size of the buffer.

Example

```
# Get the size of the buffer
print(buff.getSize())
```

ProbBuffer.resize()

Synopsis

```
resize(sz)
```

Description

Resize the size of the buffer.

Arguments

`sz`

New size of buffer.

Example

```
# Resize the size of buffer to 100
buff.resize(100)
```