GUROBI OPTIMIZER QUICK START GUIDE

Version 9.0, Copyright © 2020, Gurobi Optimization, LLC

Contents

1 Introduction **1**

2 Obtaining a Gurobi License **3**

2.1 Creating a new academic license . . . 3

3 Software Installation Guide **4**

4 Retrieving and Setting Up a Gurobi License **6**

4.1 Retrieving a Free Academic license . . . 7

Academic validation . . . 8

4.2 Retrieving a Named-User or Single-Machine or Single-Use license . . . 9

4.3 Setting up and using a Floating license . . . 11

Retrieving a Floating license . . . 11

Starting a token server . . . 13

Upgrading a token server . . . 14

Creating a token server client license . . . 15

4.4 Setting up and using a Compute Server license . . . 16

Retrieving a Compute Server license . . . 16

Creating a Compute Server client license . . . 18

4.5 Starting Gurobi Remote Services . . . 19

4.6 Using an Instant Cloud license . . . 20

4.7 Testing your license . . . 20

4.8 Setting environment variables . . . 22

5 Solving a Simple Model - The Gurobi Command Line **23**

6 Interactive Shell **28**

7 Attributes **41**

8 C Interface **42**

9 C++ Interface **49**

10 Java Interface **55**

11 .NET Interface (C#) **61**

12 Python Interface **67**

12.1 Simple Python Example 68

12.2 Python Matrix Example 72

12.3 Python Dictionary Example 76

12.4 Building and running the examples 88

13 MATLAB Interface **89**

14 R Interface **93**

15 Recommended Reading **97**

16 Installing the Anaconda Python distribution **98**

16.1 Using the Spyder IDE 100

16.2 Using Jupyter 102

17 Windows Command Line **104**

18 File Overview **106**

Introduction

Welcome to the Gurobi™ Optimizer Quick Start Guide for Windows users! This document provides a basic introduction to the Gurobi Optimizer, including:

- Information on Obtaining a Gurobi License.
- A Software Installation Guide, which includes information on Retrieving and Setting Up your License.
- An example of how to create a simple optimization model and solve it with the Gurobi Command Line, and
- A discussion of the Gurobi Interactive Shell.

We suggest that all users read these first five sections.

Once you have done this, you will probably want to choose a programming environment from which to use Gurobi. If you don't have a strong preference, we recommend that you use our Python® interface, which provides a number of benefits. First, Python is a very nice programming language that can be used for anything from experimentation to prototyping to deployment. Beyond this, though, our Python interface includes a set of higher-level modeling constructs that make it much easier to build optimization models. We also include instructions for installing the Anaconda Python distribution, which includes both a graphical development environment (Spyder) and a notebook-style interface (Jupyter).

If you already have a preferred programming language, you can select from among our available interfaces:

- C interface,
- C++ interface,
- Java® interface,
- Microsoft® .NET interface,
- Python interface,
- MATLAB® interface, or
- R interface.

At the end of the Quick Start Guide, you'll find a File Overview that lists the files included in the Gurobi distribution.

Additional resources

Once you are done with the Quick Start Guide, the next step is to explore these additional resources:

- If you are familiar with mathematical modeling and are ready to use Gurobi from one of our programming language APIs, consult the Gurobi Reference Manual.
- If you would like to see examples of ways to use Gurobi, consult the Gurobi Example Tour.
- If you are a Gurobi Compute Server user, consult the Gurobi Remote Services Reference Manual.
- If you would like to learn more about mathematical programming or modeling, we've collected a set of references in our recommended reading section.

Getting help

If you have a question that is not answered in this document, please visit the Gurobi support site at https://support.gurobi.com. There, you can read knowledge base articles and join the community discussion forum. Also, if you have a current maintenance contract, you can use the Gurobi support site to submit a request to the Gurobi support team.

Ready to get started? Your first step is to Obtain a License.

Obtaining a Gurobi License

You will need a license in order to install and use the Gurobi Optimizer. There are several ways to obtain one, depending on your situation:

- If you would like a free evaluation license, please email us at sales@gurobi.com to request one.
- If you are an academic user, you can obtain a free academic license on our website.
- If you have purchased a license from us, that license should be visible through the Current tab of your *Licenses* page on our website (you will need to login to your account to see this page).
- If you plan to use Gurobi as a client of a Gurobi Compute Server, you will need to create a compute server client license.
- If you plan to use Gurobi as a client of a Gurobi token server, you will need to create a token server client license.

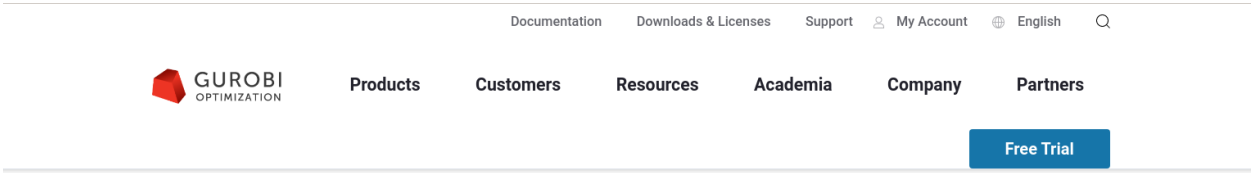
Once you have a license, your next step is to install the software.

2.1 Creating a new academic license

If you are an academic user at a degree-granting institution and wish to use an academic Gurobi license, you can create one yourself. To do so, visit the Free Academic License page on our website. You will need to read and agree to the End User License Agreement and the Conditions for academic use. Once you have done so, click on *Request License*. Your new license will be visible immediately on the Current page. You can create as many academic licenses as you like.

Your next step is to install the software.

Before using the Gurobi Optimizer, you'll need to install the software on your computer. If you haven't already done so, please go to our download page. Find your platform (64-bit Windows) and choose the corresponding file to download.



Gurobi Optimizer – Get the Software

Get the software

Gurobi Optimizer is the Gurobi optimization libraries. In addition to the software, the corresponding README file contains installation instructions. Here is the list of bug fixes for each release.

Current version		64-bit Windows	64-bit Linux	64-bit macOS	64-bit AIX
9.0.0	README	Gurobi-9.0.0-win64.msi	gurobi9.0.0_linux64.tar.gz	gurobi9.0.0_mac64.pkg	gurobi9.0.0_power64.tar.gz
md5 Checksum		17ccf7f0e1804f0a7bd5c5e70903c0b3	7878cc518522762d57ed160b3b29287a	7ff74c8f8c7265ff24c3f9c219c596e2	3a943980d36828fc8a7daa7a1b78cf28
Old versions					
8.1.1	README	Gurobi-8.1.1-win64.msi	gurobi8.1.1_linux64.tar.gz	gurobi8.1.1_mac64.pkg	gurobi8.1.1_power64.tar.gz
md5 Checksum		17dfc21f0ed64daaa4bdf7634eab705b	05cbb96072e393bd4ebb1d8b9526ce01	d05a73c0df6622851b4371dc1d292579	3d1a756695d52065eeefc15516d9aac6
8.0.1	README	Gurobi-8.0.1-win64.msi	gurobi8.0.1_linux64.tar.gz	gurobi8.0.1_mac64.pkg	gurobi8.0.1_power64.tar.gz
md5 Checksum		d9363f13daa63b79c0cdaa37ad92e8b6	cfc595ddf9482734bdc0268749093cc4	a02d04ef884e64e7091ef7a7439cfe68	877f94a02e602346ee767b9894df4030

Make a note of the name and location of the downloaded file.

Your next step is to double-click on the Gurobi installer that you downloaded from our website (e.g., `Gurobi-9.0.3-win64.msi` for the 64-bit version of Gurobi 9.0.3).

Note: if you selected *Run* when downloading you've already run the installer and don't need to do it again.

By default, the installer will place the Gurobi 9.0.3 files in directory `c:\gurobi903\win64`. The installer gives you the option to change the installation target. We'll refer to the installation directory as `<installdir>`.

Command-Line Installation

You can also install Gurobi using the command-line interface to the Windows Installer. Open a `cmd` prompt, use `cd` to go to the directory that contains the Gurobi installer image, and enter the following command:

```
msiexec /i Gurobi-9.0.3-win64.msi /norestart
```

If you are unfamiliar with running command-line commands on a Windows system, you can learn more here.

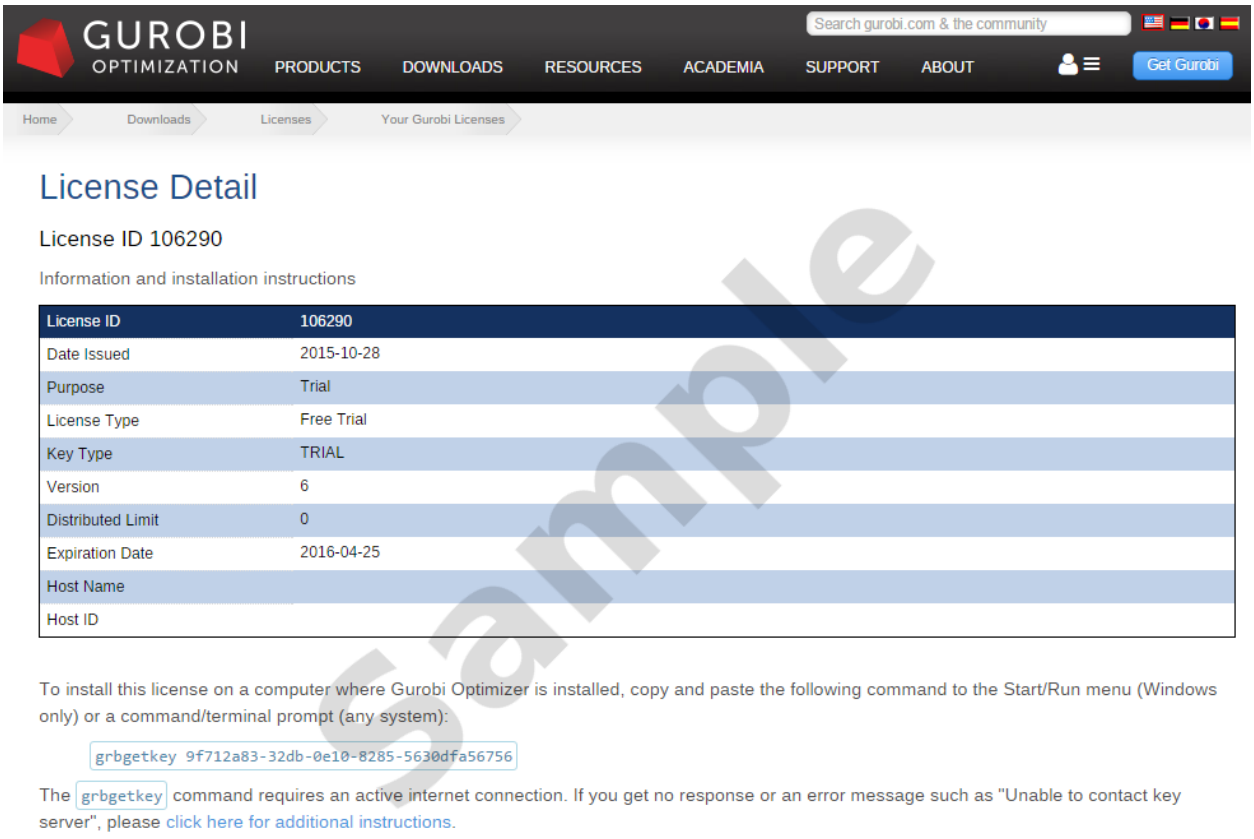
Helpful tools

To work with compressed files within the Gurobi Optimizer, we recommend that you install `gzip` (www.gzip.org) and/or `7zip` (www.7-zip.org).

You are now ready to proceed to the section on Retrieving Your Gurobi License.

If you would like an overview of the files included in the Gurobi distribution, you can also view the File Overview section.

Once your license is visible on the Current Page of the website, click on the *License ID* to view the License Detail page:



Your next step is to install this Gurobi license on your machine. You do this by obtaining a license key file.

We strongly recommend that you place your client `gurobi.lic` file in a default location for your platform (either your home directory or `c:\gurobi`). Setting up a non-default location is error-prone and a frequent source of trouble.

Please consult the `License Type` field on the License Detail page to identify your license type, and click on the appropriate link below to proceed:

- Free Academic
- Named User

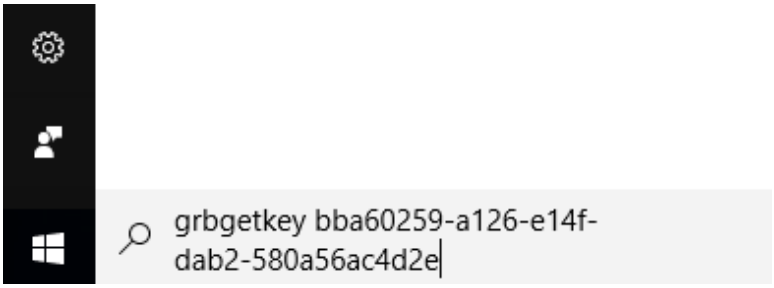
- Single-Machine
- Single-Use
- Floating
- Compute Server
- Instant Cloud

If your license includes the Distributed Add-On and you plan to use any of the Gurobi distributed algorithms, you'll also need to set up Gurobi Remote Services on your distributed worker machines.

4.1 Retrieving a Free Academic license

To obtain a Gurobi license key you'll need to run the `grbgetkey` command on your machine to retrieve your Gurobi license key. Note that the machine must be connected to the Internet in order to run this command. An Internet connection is not required after you have obtained your license key.

The exact command to run for a specific license is indicated at the bottom of the License Detail page (e.g., `grbgetkey 253e22f3-...`). We recommend that you use copy-paste to copy the entire `grbgetkey` command from our website and paste it directly into the Windows *Search* box (and then hit *Enter*):



If you are unfamiliar with running command-line commands on a Windows system, you can learn more here.

The `grbgetkey` program passes identifying information about your machine back to our website, and the website responds with your license key. Once this exchange has occurred, `grbgetkey` will ask for the name of the directory in which to store your license key file (`gurobi.lic`). You should see a prompt that looks like this:

```
Gurobi license key client (version 9.0.3)
Copyright (c) 2020, Gurobi Optimization, LLC


------------------------------
Contacting Gurobi key server...
------------------------------
```

```
Key for license ID 146542 was successfully retrieved.


---------------------
Saving license key...
---------------------


In which folderwould you like to store the Gurobi license key file?
[hit Enter to store it in c:\gurobi]:


-> License key saved to file 'c:\gurobi\gurobi.lic'.
```

You can store the license key file anywhere, but we strongly recommend that you accept the default location (either your home directory or `c:\gurobi`) by hitting *Enter*. Setting up a non-default location is error-prone and a frequent source of trouble.

If you receive an error message at this stage, it typically means that we were unable to validate your academic domain. Please consult the Academic Validation section for more information.

Using a non-default license file location

When you run the Windows version of the Gurobi Optimizer, it will look for the `gurobi.lic` key file in two different default locations: `c:\gurobi` and `c:\gurobi903` (for Gurobi 9.0.3). Note that these default paths are absolute, so for example Gurobi will look for the license key file in `c:\gurobi`, even if the software is installed in `d:\gurobi`. Note that the token server won't look for the license file in your home directory (it runs under username *LocalService*, so it doesn't have access to your home directory). If you would like to use a non-default license key file location, you can do so by setting a *system* environment variable `GRB_LICENSE_FILE` to point to the license key file. See Setting environment variables for details on how to do this.

Important note: the environment variable should point to the license key file itself, not to the directory that contains the file.

Next steps

If your license includes the Distributed Add-On and you plan to use any of the Gurobi distributed algorithms, you'll also need to set up Gurobi Remote Services on your distributed worker machines.

Once you have followed the steps above and have obtained a license key file, your next step is to test your license.

Academic validation

If you are using a free academic license, `grbgetkey` will perform an academic validation step before retrieving your license key. This step checks your domain name against our list of known academic domains. This section will help you resolve validation errors.

Not a recognized academic domain

If `grbgetkey` produces a message that looks like this:

```
ERROR 303: hostname mymachine.mydomain (234.28.234.144) not recognized as
belonging to an academic domain
```

it means your domain isn't on our academic domain list. Please make sure you are connected to your university network. If you are validating a home machine and the university provides a VPN, please connect to it before retrieving your license. If the reported host name is a valid university address, please visit our support site for assistance.

If you are having trouble validating your license through a VPN, note that some VPNs are configured to use *split tunneling*, where traffic to public internet sites is routed through your ISP. You should ask your network administrator whether the VPN can be configured to route traffic to `gurobi.com` through the private network.

Some machines connect to the internet through a proxy server. You can use the `HTTPS_PROXY` or `HTTP_PROXY` environment variables to enable `grbgetkey` to work with your proxy server. Type `grbgetkey --help` for details.

No reverse DNS information

If `grbgetkey` produces a message that only references a numerical IP address, like this:

```
ERROR 303: hostname 234.28.234.12 (234.28.234.100) not recognized as
belonging to an academic domain
```

it means your machine has no reverse DNS information. This usually happens when you are connecting to the Internet through a DHCP server that does NAT (network address translation) or PAT (port address translation), but does not provide DNS information for its clients. The simplest way to resolve this issue is to ask your network administrator to add a DNS entry (a *PTR record*) for the DHCP device itself.

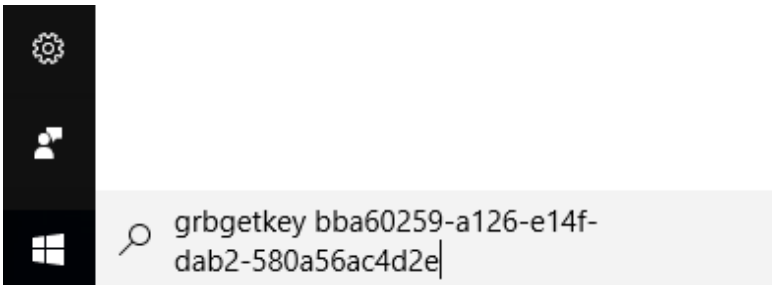
There is unfortunately no way for us to validate your academic license without reverse DNS information. You can visit this site to check DNS information for your IP address and to obtain more information about reverse DNS.

4.2 Retrieving a Named-User or Single-Machine or Single-Use license

To obtain a Gurobi license key you'll need to run the `grbgetkey` command on your machine to retrieve your Gurobi license key. Note that the machine must be connected to the Internet in order to run this command. An Internet connection is not required after you have obtained your license key.

If your computer isn't connected to the Internet or if your network security system does not allow the command below to function, we also offer a manual license key process. You'll find manual instructions at the bottom of the License Detail page (by following the link labeled *click here for additional instructions*).

The exact command to run for a specific license is indicated at the bottom of the License Detail page (e.g., `grbgetkey 253e22f3-...`). We recommend that you use copy-paste to copy the entire `grbgetkey` command from our website and paste it directly into the Windows *Search* box (and then hit *Enter*):



If you are unfamiliar with running command-line commands on a Windows system, you can learn more here.

The `grbgetkey` program passes identifying information about your machine back to our website, and the website responds with your license key. Once this exchange has occurred, `grbgetkey` will ask for the name of the directory in which to store your license key file (`gurobi.lic`). You should see a prompt that looks like this:

```
Gurobi license key client (version 9.0.3)
Copyright (c) 2020, Gurobi Optimization, LLC

------------------------------
Contacting Gurobi key server...
------------------------------

Key for license ID 146542 was successfully retrieved.

---------------------
Saving license key...
---------------------

In which folderwould you like to store the Gurobi license key file?
[hit Enter to store it in c:\gurobi]:

-> License key saved to file 'c:\gurobi\gurobi.lic'.
```

You can store the license key file anywhere, but we strongly recommend that you accept the default location (either your home directory or `c:\gurobi`) by hitting *Enter*. Setting up a non-default location is error-prone and a frequent source of trouble.

Using a non-default license file location

When you run the Windows version of the Gurobi Optimizer, it will look for the `gurobi.lic` key file in two different default locations: `c:\gurobi` and `c:\gurobi903` (for Gurobi 9.0.3). Note

that these default paths are absolute, so for example Gurobi will look for the license key file in `c:\gurobi`, even if the software is installed in `d:\gurobi`. Note that the token server won't look for the license file in your home directory (it runs under username *LocalService*, so it doesn't have access to your home directory). If you would like to use a non-default license key file location, you can do so by setting a *system* environment variable `GRB_LICENSE_FILE` to point to the license key file. See Setting environment variables for details on how to do this.

Important note: the environment variable should point to the license key file itself, not to the directory that contains the file.

Next steps

If your license includes the Distributed Add-On and you plan to use any of the Gurobi distributed algorithms, you'll also need to set up Gurobi Remote Services on your distributed worker machines.

Once you have followed the steps above and have obtained a license key file, your next step is to test your license.

4.3 Setting up and using a Floating license

When using a floating license, a program that calls the Gurobi Optimizer must obtain a token from a Gurobi token server before it can solve an optimization model. There are a few steps involved in setting up such licenses. The first is to retrieve your license key. The key should be installed on the machine that will act as your token server. Once you have your key, you will need to start the Gurobi token server. The token server is a process that runs in the background, handing out available tokens to programs as they request them. Finally, each client for the token server will need to create a token server client license to allow client programs to find the token server.

Note that if you are setting up a machine as a client of an existing token server, you just need to create a token server client license.

Retrieving a Floating license

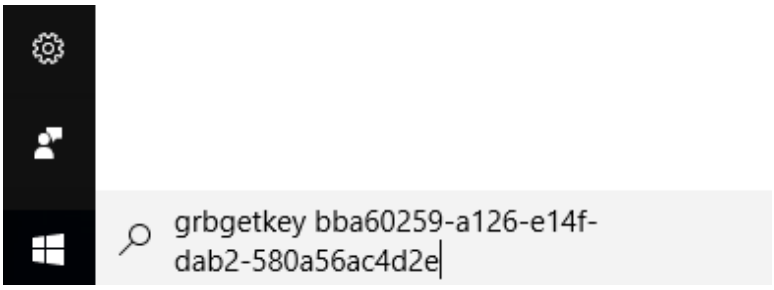
If you are using a floating license, you will need to choose a machine to act as your Gurobi token server. This token server doles out tokens to client machines. A client will request a token from the token server when it creates a Gurobi environment, and it will return the token when it destroys that environment. The client machine can be any machine that can reach the token server over your network (including the token server itself). The client can run any supported operating system. Thus, for example, a Linux client can request tokens from a Windows token server.

Once you've chosen a machine to act as your token server, you'll need to run the `grbgetkey` command on your machine to retrieve your Gurobi license key. Note that the machine must be connected to the Internet in order to run this command. An Internet connection is not required after you have obtained your license key.

If your computer isn't connected to the Internet or if your network security system does not allow the command below to function, we also offer a manual license key process. You'll find manual

instructions at the bottom of the License Detail page (by following the link labeled *click here for additional instructions*).

The exact command to run for a specific license is indicated at the bottom of the License Detail page (e.g., `grbgetkey 253e22f3-...`). We recommend that you use copy-paste to copy the entire `grbgetkey` command from our website and paste it directly into the Windows *Search* box (and then hit *Enter*):



If you are unfamiliar with running command-line commands on a Windows system, you can learn more here.

The `grbgetkey` program passes identifying information about your machine back to our website, and the website responds with your license key. Once this exchange has occurred, `grbgetkey` will ask for the name of the directory in which to store your license key file (`gurobi.lic`). You should see a prompt that looks like this:

```
Gurobi license key client (version 9.0.3)
Copyright (c) 2020, Gurobi Optimization, LLC

------------------------------
Contacting Gurobi key server...
------------------------------

Key for license ID 146542 was successfully retrieved.

---------------------
Saving license key...
---------------------

In which folderwould you like to store the Gurobi license key file?
[hit Enter to store it in c:\gurobi]:


-> License key saved to file 'c:\gurobi\gurobi.lic'.
```

You can store the license key file anywhere, but we strongly recommend that you accept the default location (either your home directory or `c:\gurobi`) by hitting *Enter*. Setting up a non-default location is error-prone and a frequent source of trouble.

Setting a password for your token server

If you want to require clients of your token server to specify a password in order to check out a token, you'll need to add one line to your `gurobi.lic` file:

```
PASSWORD=abcd1234
```

You should of course choose your own password. Clients will need to place this same line in their client license files.

When adding this line to your `gurobi.lic` file, please be careful not to modify anything else in the file.

Using a non-default license file location

When you run the Windows version of the Gurobi Optimizer, it will look for the `gurobi.lic` key file in two different default locations: `c:\gurobi` and `c:\gurobi903` (for Gurobi 9.0.3). Note that these default paths are absolute, so for example Gurobi will look for the license key file in `c:\gurobi`, even if the software is installed in `d:\gurobi`. Note that the token server won't look for the license file in your home directory (it runs under username *LocalService*, so it doesn't have access to your home directory). If you would like to use a non-default license key file location, you can do so by setting a *system* environment variable `GRB_LICENSE_FILE` to point to the license key file. See Setting environment variables for details on how to do this.

Important note: the environment variable should point to the license key file itself, not to the directory that contains the file.

Once you have followed the steps above and have obtained a license key file, your next step is to start the token server.

Starting a token server

Important note: most Gurobi licenses do not use the token server. You should only follow these instructions if you are setting up a floating license. If you are not sure whether you need to start a token server, you can examine the contents of your `gurobi.lic` file. If it contains the line `TYPE=TOKEN`, and does not contain the line `MACHINELIMIT=0`, then you need a token server.

On a Windows system, you can start the token service by selecting the *Gurobi Token Server* menu item from the *Gurobi* folder of the *Start* menu. You should only do so after you have installed the Gurobi license key file.

By default, the token server only produces logging output when it starts. To obtain more detailed logging information, start the token server with the `-v` switch. This will produce a log message each time a token is checked in or out.

The token server runs as a Windows service. If you'd like to run it in the foreground, start it from and command window and use the `-n` switch.

Firewalls

The next step after starting the Gurobi token server depends on your anti-virus software and firewall settings. Most anti-virus software will immediately ask you to confirm that you are allowing program `grb_ts.exe` to receive network traffic. Once you confirm this, the token server will start serving tokens. If you don't receive such a prompt, you will need to add `grb_ts.exe` to the firewall exceptions list. To do this, select *Allow a program through Windows firewall* under the *Security* area of the Control Panel (labeled *Allow an app through Windows firewall* in Windows 8).

Some machines have more restrictive firewalls that may require additional action. The Gurobi token server uses port 41954 by default. If you are unable to reach the token server after taking the steps described, you should ask your network administrator for more information on how to open the required port.

Starting and stopping the grb_ts Windows service

Once the token service has been started, you should see the `grb_ts` service listed in the *Services* tab of the Task Manager. To start or stop the service, click on the *Services* button at the bottom-right of the *Services* tab, and then right-click on the *Gurobi Token Server* item on this screen.

You can also start or stop the Gurobi Token Server service from a Console window with administrator privileges. Running `grb_ts -h` lists command-line options. Issuing a `grb_ts -s` command stops a running token server. If you are unfamiliar with running command-line commands on a Windows system, you can learn more here.

Add the line

```
VERBOSE=1
```

to your `gurobi.lic` file to start the license service in verbose mode. Verbose mode produces a log message (in the Windows Event Log) each time a token is checked in or out.

Next steps

Clients of the token server also need simple license files. Your next step is to set up a client license.

Once your token server is running and you've set up a client license, you can move on to testing the license.

Once you've set up a client license, you can test the state of the token server at any time, as well as get a list of the clients that are currently using tokens, by typing `gurobi_cl --tokens`.

Upgrading a token server

To upgrade your token server from an earlier version of the Gurobi Optimizer, you will need to perform the following steps (on the machine running the token server):

1. Stop the old token server.
2. Install the new version of the Gurobi Optimizer.

3. Upgrade your license file (or modify `GRB_LICENSE_FILE` to point to the new license file).

4. Start the new token server.

Creating a token server client license

The purpose of a token server client license is quite simple: it tells the client where to find the Gurobi token server. You can create this file yourself (using a text editor like `WordPad`, for example). The client `gurobi.lic` file typically contains a single line of text:

```
TOKENSERVER=mymachine.mydomain.com
```

or:

```
TOKENSERVER=192.168.1.100
```

You should of course substitute the name or IP address of your token server in the example above.

If your token server was configured to use a non-default port, you'll also need a line that provides the port number:

```
PORT=46325
```

The client license file may also include four optional lines. A `QUEUETIMEOUT` line allows you to set a limit (in seconds) on how long a new Compute Server job will wait in queue before it gives up (and reports a `JOB_REJECTED` error). Any negative value will allow a job to sit in the Compute Server queue indefinitely.

An `IDLETIMEOUT` line allows you to set a limit on how long a Compute Server job can sit idle before the server kills the job (in seconds). A job is considered idle if the server is not currently performing an optimization and the client has not issued any additional commands. The default value will allow a job to sit idle indefinitely in all but a few circumstances. The first exception is the Gurobi Instant Cloud, where the default setting will automatically impose a 30 minute idle time limit (1800 seconds). If you are using an Instant Cloud pool, the actual value will be the maximum between this parameter value and the idle timeout defined by the pool. The second exception is any program that uses the Gurobi Python interface (including the Gurobi Interactive Shell). Such programs will also get a 30 minute idle time limit by default.

A `SERVERTIMEOUT` line allows you to specify the timeout (in seconds) in case the token server is unavailable. The default value is 30 seconds. If the client program is unable to contact the server for more than the specified amount of time, the client will quit with a network error.

A `PASSWORD` line allows you to connect to a password-protected token server (you'll need to get the password from the owner of the token server).

A more complex client token file might look like this:

```
TOKENSERVER=192.168.1.100
IDLETIMEOUT=60
QUEUETIMEOUT=120
SERVERTIMEOUT=10
PASSWORD=abcd1234
```

We strongly recommend that you place your client `gurobi.lic` file in a default location for your platform (either your home directory or `c:\gurobi`). Setting up a non-default location is error-prone and a frequent source of trouble. (If you still want to use a non-default location, please refer to the instructions that appeared earlier in this section).

If your client and the token server are both running on the same machine, they can share a single `gurobi.lic` file. You just need to add the following line to the `gurobi.lic` file you obtained from our website:

```
TOKENSERVER=localhost
```

The token server will ignore this line, and the client will ignore everything but this line. Your other option when both client and server are running on the same machine is to create a separate `gurobi.lic` file for the client, and to set the `GRB_LICENSE_FILE` environment variable to point to this file (following the earlier instructions for using a non-default license location).

Once your client license is in place, you can test the license. If you are unable to connect to the server, you'll need to make sure the server is installed and running. Please consult the instructions for starting a token server for more information.

4.4 Setting up and using a Compute Server license

When using a Compute Server license, programs that call the Gurobi Optimizer can offload Gurobi computations onto one or more machines. There are a few steps involved in setting up such licenses. The first is to retrieve your license key. The key should be installed on the machine that will act as a Compute Server. Once you have your key, you will need to start Gurobi Remote Services. Finally, client machines will need a Compute Server client license in order to find the Compute Server(s).

Note that if you are setting up a machine as a client of an existing Compute Server, you just need to create a Compute Server client license.

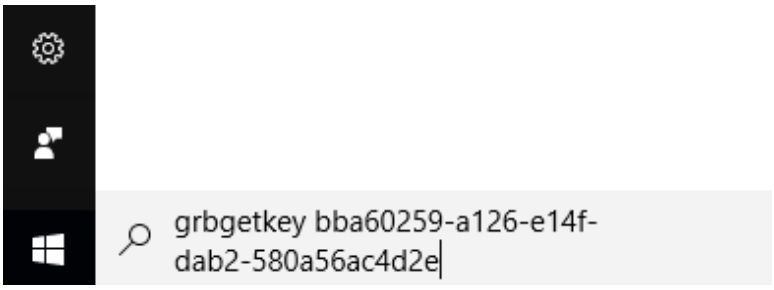
Retrieving a Compute Server license

If you have purchased one or more Gurobi Compute Server licenses, you'll need to perform a few setup steps in order to start your Compute Servers. Once started, client machines will be able to offload the work of solving an optimization model onto these servers. The clients and the Compute Servers can run any mix of supported operating systems. Thus, for example, multiple Linux machines could submit jobs to a pair of Compute Servers, one running Windows and the other running Linux. Any machine that can reach the Compute Server(s) over your network can be a client (including the Compute Servers themselves).

Once you've chosen a machine to act as a Compute Server (or a node in a Compute Server cluster), you'll need to run the `grbgetkey` command on your machine to retrieve your Gurobi license key. Note that the machine must be connected to the Internet in order to run this command. An Internet connection is not required after you have obtained your license key.

If your computer isn't connected to the Internet or if your network security system does not allow the command below to function, we also offer a manual license key process. You'll find manual instructions at the bottom of the License Detail page (by following the link labeled *click here for additional instructions*).

The exact command to run for a specific license is indicated at the bottom of the License Detail page (e.g., `grbgetkey 253e22f3-...`). We recommend that you use copy-paste to copy the entire `grbgetkey` command from our website and paste it directly into the Windows *Search* box (and then hit *Enter*):



If you are unfamiliar with running command-line commands on a Windows system, you can learn more here.

The `grbgetkey` program passes identifying information about your machine back to our website, and the website responds with your license key. Once this exchange has occurred, `grbgetkey` will ask for the name of the directory in which to store your license key file (`gurobi.lic`). You should see a prompt that looks like this:

```
Gurobi license key client (version 9.0.3)
Copyright (c) 2020, Gurobi Optimization, LLC

------------------------------
Contacting Gurobi key server...
------------------------------

Key for license ID 146542 was successfully retrieved.

---------------------
Saving license key...
---------------------

In which folderwould you like to store the Gurobi license key file?
[hit Enter to store it in c:\gurobi]:

-> License key saved to file 'c:\gurobi\gurobi.lic'.
```

You can store the license key file anywhere, but we strongly recommend that you accept the default location (either your home directory or `c:\gurobi`) by hitting *Enter*. Setting up a non-default location is error-prone and a frequent source of trouble.

Using a non-default license file location

When you run the Windows version of the Gurobi Optimizer, it will look for the `gurobi.lic` key file in two different default locations: `c:\gurobi` and `c:\gurobi903` (for Gurobi 9.0.3). Note that these default paths are absolute, so for example Gurobi will look for the license key file in `c:\gurobi`, even if the software is installed in `d:\gurobi`. Note that the token server won't look for the license file in your home directory (it runs under username *LocalService*, so it doesn't have access to your home directory). If you would like to use a non-default license key file location, you can do so by setting a *system* environment variable `GRB_LICENSE_FILE` to point to the license key file. See Setting environment variables for details on how to do this.

Important note: the environment variable should point to the license key file itself, not to the directory that contains the file.

Once you have followed the steps above and have obtained a license key file, your next step is to start Gurobi Remote Services.

Creating a Compute Server client license

If you are a Compute Server user, we recommend that you read the Gurobi Remote Services Reference Manual for information about configuring and using Remote Services. We'll provide a few relevant details here, but this other document provides a much broader overview.

You have two options for indicating that a Gurobi program will act as a client of a Compute Server. If you are writing a program that calls the Gurobi C, C++, Java, .NET, or Python APIs, these APIs provide routines that allow you to specify the name of a Compute Server node (by creating an `empty environment` and then setting parameters related to Compute Server on that environment). If you use these routines, Gurobi licenses aren't required on the client.

Alternately, you can set up a `gurobi.lic` file that points to the Compute Server. This option allows you to use a Compute Server with nearly any program that calls Gurobi, without the need to modify the calling program. You can create your client `gurobi.lic` with a text editor like `WordPad`. The file should contain a line that looks like this:

```
COMPUTESERVER=server.mydomain.com:61000
```

or like this:

```
COMPUTESERVER=192.168.1.100:61000
```

This line should provide the name or IP address of any machine in your Compute Server cluster, optionally followed by the chosen port number on that server (which was chosen when you set up the Compute Server on that machine). If your Compute Server uses a password, you should also include a line that gives the password:

```
PASSWORD=cspwd
```

Please consult the *Using Remote Services* section of the Gurobi Remote Services Reference Manual for more information.

Note that if your client and server are both running on the same machine, you'll need to create a separate `gurobi.lic` file for the client, and set the `GRB_LICENSE_FILE` environment variable to point to this file (following the earlier instructions for using a non-default license location).

Once your client license is in place, you can test the license. If you are unable to connect to the server, you'll need to make sure the server is installed and running. Please consult the *Cluster Setup and Administration* section of the Gurobi Remote Services Reference Manual for more information.

4.5 Starting Gurobi Remote Services

Important note: you only need to start Gurobi Remote Services if you are setting up a Compute Server or a distributed worker (for use in distributed algorithms). If you are not sure whether you need to start Gurobi Remote Services, you can examine the contents of your `gurobi.lic` file. If it contains the line `CSENABLED=1`, then you need Gurobi Remote Services. If it contains a line that begins with `DISTRIBUTED=`, and if you plan to run distributed algorithms, then you also need Gurobi Remote Services.

Gurobi Remote Services is a Windows Service that allows one or more machines to perform Gurobi computations on behalf of other client machines. The set of services a server can provide will depend on your license. If you are setting up a machine as a distributed worker, no license is required. In this case, the only service provided by the server is to act as a worker in a distributed algorithm. If you have a Compute Server license, then servers running Gurobi Remote Services can provide a variety of services, including offloading computation from a set of clients, balancing computational load among the servers, and providing failover capabilities, in addition to acting as a distributed worker. In this case, be sure that your license key file has been installed before starting Gurobi Remote Services.

You'll find instructions for installing, starting, and stopping Gurobi Remote Services in the *Cluster Setup and Administration* section of the Gurobi Remote Services Reference Manual.

Note that Gurobi Remote Services is distributed as a separate installer (named gurobi_server8.0.0.msion the Windowsplatform). You'll need to download that file separately from our download page.

Next steps

Once you've set up Gurobi Remote Services, you should test the state of the server. Type this command on your server (assuming you have configured your Compute Server to use port 61000):

```
gurobi_cl --server=localhost:61000
```

If the output includes the following line:

```
Capacity available on 'localhost:61000' - connecting...
```

then Remote Services is ready for use.

Client programs will need to know how to reach your server. If you are using Gurobi Compute Server, this is typically done with a client license file. You should set that up now.

4.6 Using an Instant Cloud license

You have two options for indicating that a Gurobi program will act as a client of a Gurobi Instant Cloud instance. If you are writing a program that calls the Gurobi C, C++, Java, .NET, or Python APIs, these APIs provide routines that allow you to launch cloud instances (by creating an `empty environment` and then setting parameters related to the cloud on that environment). If you use these routines, Gurobi licenses aren't required on the client.

Alternately, you can install a `gurobi.lic` file that points to your Gurobi Instant Cloud. This option allows you to use Instant Cloud with nearly any program that calls Gurobi, without the need to modify the calling program. You can download a client `gurobi.lic` file from your account on the Instant Cloud website. You should place this file in a default location (either your home directory or `c:\gurobi`), or set the `GRB_LICENSE_FILE` environment variable to point to it.

This file will contain two lines that look like this:

```
CLOUDACCESSID=312e9gef-e0bc-4114-b6fb-26ed7klaeff9
CLOUDKEY=ae32L0H321dgaL
```

If you are using a non-default pool, the file should also indicate the name of that pool:

```
CLOUDPOOL=myPool
```

These lines allow your client program to launch and use cloud instances from your account. You should keep this information private, since anyone with access to it can also use your cloud instances.

Please consult the *Gurobi Instant Cloud* section of the Gurobi Remote Services Reference Manual for more information.

4.7 Testing your license

Once you have obtained a license key for your machine, you are ready to test your license using the Gurobi Interactive Shell. To do this, double-click on the Gurobi icon on your desktop. The shell should produce the following output:

```
Using license file c:\gurobi\gurobi.lic
Set parameter LogFile to value gurobi.log

Gurobi Interactive Shell, Version 9.0.3
Copyright (c) 2020, Gurobi Optimization, LLC
Type "help()" for help


gurobi>
```

If you are running as a client of a Gurobi Compute Server, the message above will be preceded by a message like this:

```
Capacity available on 'myserver' - connecting...
Established HTTP unencrypted connection
```

Congratulations, your license is functioning correctly! You are now ready to use the Gurobi Optimizer. The next section will show you how to solve a simple optimization model.

Possible errors

If the Gurobi shell didn't produce the desired output, there's a problem with your license. We'll list a few common errors here.

The following message...

```
ERROR: No Gurobi license found (user smith, host mymachine, hostid 9d3128ce)
```

indicates that your `gurobi.lic` file couldn't be found.

Did you use a non-default license file location? When you run the Windows version of the Gurobi Optimizer, it will look for the `gurobi.lic` key file in three different default locations. It will always look in your home directory. In addition, Gurobi Optimizer 9.0.3 will also look in `c:\gurobi` and `c:\gurobi903`. Note that these default paths are absolute, so for example Gurobi will look for the license key file in `c:\gurobi`, even if the software is installed in `d:\gurobi`. If you used a non-default license key file location, you should set environment variable `GRB_LICENSE_FILE` to point to the license key file. See Setting environment variables for details on how to do this.

Important note: the environment variable should point to the license key file itself, not to the directory that contains the file.

The following message:

```
ERROR: HostID mismatch (licensed to 9d3128ce, hostid is 7de025e9)
```

indicates that your `gurobi.lic` isn't valid for this machine. You should make sure that you are using the right `gurobi.lic` file.

If you are running as a client of a Gurobi token server and receive this message:

```
ERROR: Failed to connect to token server 'myserver' (port 41954)
```

the token server isn't currently running. If you receive this message:

```
ERROR: No TOKENSERVER specified for TOKEN license
```

your license file is missing the `TOKENSERVER=` line that provides the name of your token server. Please consult the section on setting up a token server.

If you are running as a client of a Gurobi Compute Server and receive this message:

```
ERROR: No server available
```

the Compute Server isn't currently running. Please consult the section on setting up a Compute Server.

If, after following the instructions, you still experience problems while setting up or testing your license, please visit our support site for assistance.

4.8 Setting environment variables

Gurobi uses system variables for multiple configuration purposes. For example, to specify where to look for a license file, you must set the variable `GRB_LICENSE_FILE`.

On Windows systems, environment variables are created and modified through the Control Panel. Searching for *Environment Variables* from the Control Panel search box will lead you to the appropriate screen. You will need to add a new *System variable* named `GRB_LICENSE_FILE`, and set it to the location of your license file (e.g., `d:\gurobi\gurobi.lic`). *Important note: your new environment variable must be a System variable, not a User variable.*

Solving a Simple Model - The Gurobi Command Line

Now that the Gurobi Optimizer is installed and the license key has been tested, you're ready to solve a simple math programming model.

This section includes instructions on how to create a simple math programming model and how to use the Gurobi command-line interface to compute an optimal solution. If you are already familiar with mathematical modeling and LP-format files, feel free to skip to the end of this section.

Note that this section gives only a brief glimpse into the capabilities of the Gurobi command-line interface. This interface plays a number of different roles. In addition to providing a simple interface for solving a model, it can also be used to launch a model on a Compute Server or on a cloud system, and it can check on the status of a token server. If you'd like additional information, consult the Command-Line Tool section of our Reference Manual.

The problem statement - producing coins

We'll begin by stating the problem to be solved.

Imagine that it is the end of the calendar year at the United States Mint. The Mint keeps an inventory of the various minerals used to produce the coins that are put into circulation, and it wants to use up the minerals on hand before retooling for next year's coins.

The Mint produces several different types of coins, each with a different composition. The table below shows the make-up of each coin type (as reported in the US Mint coin specifications).

	Penny	Nickel	Dime	Quarter	Dollar
Copper (Cu)	0.06g	3.8g	2.1g	5.2g	7.2g
Nickel (Ni)		1.2g	0.2g	0.5g	0.2g
Zinc (Zi)	2.4g				0.5g
Manganese (Mn)					0.3g

Suppose the Mint wants to use the available materials to produce coins with the maximum total dollar value. Which coins should they produce?

The optimization model

In order to formulate this as an optimization problem, we'll need to do three things.

- First, we'll need to define the decision variables. The goal of the optimization is to choose values for these variables.

- Second, we'll define a linear objective function. This is the function we'd like to minimize (or maximize).
- Third, we'll define the linear constraints.

The Gurobi Optimizer will consider all assignments of values to decision variables that satisfy the specified linear constraints, and return one that optimizes the stated objective function.

The variables in this problem are quite straightforward. The solver will need to decide how many of each coin to produce. It is convenient to give the decision variables meaningful names. In this case, we'll call the variables *Pennies*, *Nickels*, *Dimes*, *Quarters*, and *Dollars*. We'll also introduce variables that capture the quantities of the various minerals actually used by the solution. We'll call them *Cu*, *Ni*, *Zi*, and *Mn*.

Recall that the objective of our optimization problem is to maximize the total dollar value of the coins produced. Each penny produced is worth 0.01 dollars, each nickel is worth 0.05 dollars, etc. This gives the following linear objective:

```
maximize: 0.01 Pennies + 0.05 Nickels + 0.1 Dimes + 0.25 Quarters + 1 Dollars
```

The constraints of this model come from the fact that producing a coin consumes certain quantities of the available minerals, and the supplies of those minerals are limited. We'll capture these relationships in two parts. First, we'll create an equation for each mineral that captures the amount of that mineral that is consumed. For copper, that equation would be:

```
Cu = 0.06 Pennies + 3.8 Nickels + 2.1 Dimes + 5.2 Quarters + 7.2 Dollars
```

The coefficients for this equation come from the earlier coin specification table: one penny uses 0.06g of copper, one nickel uses 3.8g, etc.

The model must also capture the available quantities of each mineral. If we have 1000 grams of copper available, then the constraint would be:

```
Cu <= 1000
```

For our example, we'll assume we have 1000 grams of copper and 50 grams of the other minerals.

There is actually one other set of constraints that must be captured in order for our model to accurately reflect the physical realities of our problem. While a dime is worth 10 cents, half of a dime isn't worth 5 cents. The variables that capture the number of each coin produced must take integer values.

The model file

The Gurobi Optimizer provides a variety of options for expressing an optimization model. Typically, you would build the model using an interface to a programming languages (C, C++, C#, Java, etc.) or using a higher-level application environment (a notebook, spreadsheet, a modeling system, MATLAB, R, etc.). However, to keep our example as simple as possible, we're going to read the model from an LP format file. The LP format was designed to be human readable, and as such it is well suited for our needs.

The LP format is mostly self-explanatory. Here is our model:

```
Maximize
  .01 Pennies + .05 Nickels + .1 Dimes + .25 Quarters + 1 Dollars
Subject To
  Copper: .06 Pennies + 3.8 Nickels + 2.1 Dimes + 5.2 Quarters + 7.2 Dollars -
     Cu = 0
  Nickel: 1.2 Nickels + .2 Dimes + .5 Quarters + .2 Dollars -
     Ni = 0
  Zinc: 2.4 Pennies + .5 Dollars - Zi = 0
  Manganese: .3 Dollars - Mn = 0
Bounds
  Cu <= 1000
  Ni <= 50
  Zi <= 50
  Mn <= 50
Integers
  Pennies Nickels Dimes Quarters Dollars
End
```

You'll find this model in file `coins.lp` in the `<installdir>/examples/data` directory of your Gurobi distribution. Specifically, assuming you've installed Gurobi 9.0.3 in the recommended location, you'll find the file in `c:\gurobi903\win64\examples\data\coins.lp`.

To modify this file, open it in a text editor (like `WordPad`).

Editing the LP file

Before you consider making any modifications to this file or creating your own, we should point out a few rules about LP format files.

1. Ordering of the sections

Our example contains an objective section (`Maximize...`), a constraint section (`Subject To...`), a variable bound section (`Bounds...`), and an integrality section (`Integers...`). The sections must come in that order. The complete list of section types and the associated ordering rules can be found in the file format section of the Gurobi Reference Manual.

2. Separating tokens

Tokens must be separated by either a space or a newline. Thus, for example, the term:

```
+ .1 Dimes
```

must include a space or newline between `+` and `.1`, and another between `.1` and `Dimes`.

3. Arranging variables

Variables must always appear on the left-hand side of a constraint. The right-hand side is always a constant. Thus, our constraint:

```
  Cu = .06 Pennies + 3.8 Nickels + 2.1 Dimes + 5.2 Quarters + 7.2 Dollars
```

...becomes...

```
  .06 Pennies + 3.8 Nickels + 2.1 Dimes + 5.2 Quarters + 7.2 Dollars - Cu = 0
```

4. Variable default bounds

Unless stated otherwise, a variable has a zero lower bound and an infinite upper bound. Thus, `Cu <= 1000` really means `0 <= Cu <= 1000`. Similarly, any variable not mentioned in the `Bounds` section may take any non-negative value.

Full details on the LP file format are provided in the file format section of the Gurobi Reference Manual.

Solving the model using the Gurobi command-line interface

The final step in solving our optimization problem is to pass the model to the Gurobi Optimizer. We'll use the Gurobi command-line interface, as it is typically the simplest of our interfaces to use when solving a model stored in a file.

To use the command-line interface, you'll first need to open a Console window. If you are unfamiliar with running command-line commands on a Windows system, you can learn more here. (Note that the Gurobi Interactive Shell, which was used earlier to test your license, does *not* directly accept command-line program input).

The name of the Gurobi command-line tool is `gurobi_cl`. To invoke it, type `gurobi_cl`, followed by the name of the model file. For example, if our model is stored in the file `c:\gurobi903\win64\examples\data\coins.lp`, you would type the following command into your command-line window...

```
>gurobi_cl c:\gurobi903\win64\examples\data\coins.lp
```

This command should produce the following output...

```
Using license file c:\gurobi\gurobi.lic
Set parameter LogFile to value gurobi.log

Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (linux64)
Copyright (c) 2020, Gurobi Optimization, LLC

Read LP format model from file c:/gurobi903/win64/examples/data/coins.lp
Reading time = 0.00 seconds
: 4 rows, 9 columns, 16 nonzeros
Optimize a model with 4 rows, 9 columns and 16 nonzeros
Model fingerprint: 0xa0c5449c
Variable types: 4 continuous, 5 integer (0 binary)
Coefficient statistics:
  Matrix range     [6e-02, 7e+00]
  Objective range  [1e-02, 1e+00]
  Bounds range     [5e+01, 1e+03]
  RHS range        [0e+00, 0e+00]
Found heuristic solution: objective -0.0000000
Presolve removed 1 rows and 5 columns
Presolve time: 0.00s
Presolved: 3 rows, 4 columns, 9 nonzeros
Variable types: 0 continuous, 4 integer (0 binary)

Root relaxation: objective 1.134615e+02, 2 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
```

```
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  113.46154    0    1   -0.00000  113.46154      -     -    0s
H    0     0                       113.4500000  113.46154  0.01%     -    0s
     0     0  113.46154    0    1  113.45000  113.46154  0.01%     -    0s

Explored 1 nodes (2 simplex iterations) in 0.00 seconds
Thread count was 8 (of 8 available processors)

Solution count 2: 113.45 -0

Optimal solution found (tolerance 1.00e-04)
Best objective 1.134500000000e+02, best bound 1.134500000000e+02, gap 0.0000%
```

Details on the format of the Gurobi log file can be found in the Gurobi Reference Manual. For now, you can simply note that the optimal objective value is 113.45. Recall that the objective is denoted in dollars. We can therefore conclude that by a proper choice of production plan, the Mint can produce $113.45 worth of coins using the available minerals. Moreover, because this value is optimal, we know that it is not possible to produce coins with value greater than $113.45!

It would clearly be useful to know the exact number of each coin produced by this optimal plan. The `gurobi_cl` command allows you to set Gurobi parameters through command-line arguments. One particularly useful parameter for the purposes of this example is `ResultFile`, which instructs the Gurobi Optimizer to write a file once optimization is complete. The type of the file is encoded in the suffix. To request a `.sol` file:

```
> gurobi_cl ResultFile=coins.sol coins.lp
```

The command will produce a file that contains solution values for the variables in the model:

```
# Objective value = 113.45
Pennies 0
Nickels 0
Dimes 2
Quarters 53
Dollars 100
Cu 999.8
Ni 46.9
Zi 50
Mn 30
```

In the optimal solution, we'll produce 100 dollar coins, 53 quarters, and 2 dimes.

If we wanted to explore the parameters of the model (for example, to consider how the optimal solution changes with different quantities of available minerals), we could use a text editor to modify the input file. However, it is typically better to do such tests within a more powerful system. We'll now describe the Gurobi Interactive Shell, which provides an environment for creating, modifying, and experimenting with optimization models.

Interactive Shell

The Gurobi interactive shell allows you to perform hands-on interaction and experimentation with optimization models. You can read models from files, perform complete or partial optimization runs on them, change parameters, modify the models, reoptimize, and so on. The Gurobi shell is actually a set of extensions to the Python shell. Python is a rich and flexible programming language, and any capabilities that are available from Python are also available from the Gurobi shell. We'll touch on these capabilities here, but we encourage you to explore the help system and experiment with the interface in order to gain a better understanding of what is possible.

One big advantage of working within Python is that the Python language is popular and well supported. One aspect of this support is the breadth of powerful Python Integrated Development Environments (IDEs) that are available, most of which can be downloaded for free from the internet. This document includes instructions for setting up Gurobi for use within the Anaconda distribution. Anaconda includes a powerful IDE (Spyder), as well as a notebook-style interface (Jupyter).

Before diving into the details of the Gurobi interactive shell, we should remind you that Gurobi also provides a lightweight command line interface. If you just need to do a quick test on a model stored in a file, you will probably find that that interface is better suited to simple tasks than the interactive shell.

Important note for AIX users: due to limited Python support on AIX, our AIX port does not include the Interactive Shell or the Python interface. You can use the command line, or the C, C++, or Java interfaces.

We will now work through a few simple examples of how the Gurobi shell might be used, in order to give you a quick introduction to its capabilities. More thorough documentation on this and other interfaces is available in the Gurobi Reference Manual.

Reading and optimizing a model

There are several ways to access the Gurobi Interactive Shell.

From Windows, you can:

- Double-click on the Gurobi desktop shortcut.
- Select the Gurobi Interactive Shell from the Start Menu.
- Open a DOS command shell and type `gurobi.bat`.

If you've installed a Python IDE, the shell will also be available from that environment.

Once the optimizer has started, you are ready to load and optimize a model. We'll consider model `coins.lp` from `<installdir>/examples/data`...

```
> gurobi.bat

Using license file c:\gurobi\gurobi.lic
Set parameter LogFile to value gurobi.log

Gurobi Interactive Shell, Version 9.0.3
Copyright (c) 2020, Gurobi Optimization, LLC
Type "help()" for help

gurobi> m = read('c:/gurobi903/win64/examples/data/coins.lp')
Read LP format model from file c:/gurobi903/win64/examples/data/coins.lp
Reading time = 0.01 seconds
: 4 rows, 9 columns, 16 nonzeros
gurobi> m.optimize()
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 4 rows, 9 columns and 16 nonzeros
Model fingerprint: 0xa0c5449c
Variable types: 4 continuous, 5 integer (0 binary)
Coefficient statistics:
  Matrix range     [6e-02, 7e+00]
  Objective range  [1e-02, 1e+00]
  Bounds range     [5e+01, 1e+03]
  RHS range        [0e+00, 0e+00]
Found heuristic solution: objective -0.0000000
Presolve removed 1 rows and 5 columns
Presolve time: 0.00s
Presolved: 3 rows, 4 columns, 9 nonzeros
Variable types: 0 continuous, 4 integer (0 binary)

Root relaxation: objective 1.134615e+02, 2 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  113.46154    0    1   -0.00000  113.46154      -     -    0s
H    0     0                       113.4500000  113.46154  0.01%     -    0s
     0     0  113.46154    0    1  113.45000  113.46154  0.01%     -    0s

Explored 1 nodes (2 simplex iterations) in 0.01 seconds
Thread count was 8 (of 8 available processors)

Solution count 2: 113.45 -0

Optimal solution found (tolerance 1.00e-04)
Best objective 1.134500000000e+02, best bound 1.134500000000e+02, gap 0.0000%
```

The *read()* command reads a model from a file and returns a `Model` object. In the example, that object is placed into variable `m`. There is no need to declare variables in the Python language; you simply assign a value to a variable.

Note that *read()* accepts wildcard characters, so you could also have said:

```
gurobi> m = read('c:/gurobi903/win64/*/*/coin*')
```

Note also that Gurobi commands that read or write files will also function correctly with compressed files. If `gzip`, `bzip2`, or `7zip` are installed on your machine and available in your default path, then

you simply need to add the appropriate suffix (`.gz`, `.bz2`, `.zip`, or `.7z`) to the file name to read or write compressed versions.

The next statement in the example, *m.optimize()*, invokes the `optimize` method on the `Model` object (you can obtain a list of all methods on `Model` objects by typing `help(Model)` or `help(m)`). The Gurobi optimization engine finds an optimal solution with objective 113.45.

Inspecting the solution

Once a model has been solved, you can inspect the values of the model variables in the optimal solution with the *printAttr()* method on the `Model` object:

```
gurobi> m.printAttr('X')
    Variable            X
-------------------------
       Dimes            2
    Quarters           53
     Dollars          100
          Cu        999.8
          Ni         46.9
          Zi           50
          Mn           30
```

This routine prints all non-zero values of the specified attribute `X`. Attributes play a major role in the Gurobi optimizer. We'll discuss them in more detail shortly.

You can also inspect the results of the optimization at a finer grain by retrieving a list of all the variables in the model using the *getVars()* method on the `Model` object (*m.getVars()* in our example):

```
gurobi> v = m.getVars()
gurobi> print(len(v))
9
```

The first command assigns the list of all `Var` objects in model `m` to variable `v`. The Python `len()` command gives the length of the array (our example model `coins` has 9 variables). You can then query various attributes of the individual variables in the list. For example, to obtain the variable name and solution value for the first variable in list `v`, you would issue the following command:

```
gurobi> print(v[0].varName, v[0].x)
Pennies 0.0
```

You can type `help(Var)` or `help(v[0])` to get a list of all methods on a `Var` object. You can type `help(GRB.Attr)` to get a list of all attributes.

Simple model modification

We will now demonstrate a simple model modification. Imagine that you only want to consider solutions where you make at least 10 pennies (i.e., where the *Pennies* variable has a lower bound of 10.0). This is done by setting the `lb` attribute on the appropriate variable (the first variable in the list `v` in our example)...

```
gurobi> v = m.getVars()
gurobi> v[0].lb = 10
```

The Gurobi optimizer keeps track of the state of the model, so it knows that the currently loaded solution is not necessarily optimal for the modified model. When you invoke the *optimize()* method again, it performs a new optimization on the modified model...

```
gurobi> m.optimize()
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)

Optimize a model with 4 rows, 9 columns and 16 nonzeros
Variable types: 4 continuous, 5 integer (0 binary)
Coefficient statistics:
  Matrix range     [6e-02, 7e+00]
  Objective range  [1e-02, 1e+00]
  Bounds range     [1e+01, 1e+03]
  RHS range        [0e+00, 0e+00]
Presolve removed 2 rows and 5 columns
Presolve time: 0.00s
Presolved: 2 rows, 4 columns, 5 nonzeros

MIP start from previous solve did not produce a new incumbent solution

Variable types: 0 continuous, 4 integer (0 binary)
Found heuristic solution: objective 25.9500000

Root relaxation: objective 7.190000e+01, 2 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0   71.90000    0    1   25.95000   71.90000   177%     -    0s
H    0     0                      71.8500000   71.90000  0.07%     -    0s
H    0     0                      71.9000000   71.90000  0.00%     -    0s

Explored 0 nodes (2 simplex iterations) in 0.01 seconds
Thread count was 8 (of 8 available processors)

Solution count 3: 71.9 71.85 25.95

Optimal solution found (tolerance 1.00e-04)
Best objective 7.190000000000e+01, best bound 7.190000000000e+01, gap 0.0000%
```

The result shows that, if you force the solution to include at least 10 pennies, the maximum possible objective value for the model decreases from 113.45 to 71.9. A simple check confirms that the new lower bound is respected:

```
gurobi> print(v[0].x)
10.0
```

Simple experimentation with a more difficult model

Let us now consider a more difficult model, `glass4.mps`. Again, we read the model and begin the optimization:

```
gurobi> m = read('c:/gurobi903/win64/examples/data/glass4')
Read MPS format model from file c:/gurobi903/win64/examples/data/glass4.mps
Reading time = 0.00 seconds
glass4:  396 Rows, 322 Columns, 1815 NonZeros
gurobi> m.optimize()
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)

Optimize a model with 396 rows, 322 columns and 1815 nonzeros
Model fingerprint: 0x541d0ad3
Variable types: 20 continuous, 302 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 8e+06]
  Objective range  [1e+00, 1e+06]
  Bounds range     [1e+00, 8e+02]
  RHS range        [1e+00, 8e+06]
Presolve removed 4 rows and 5 columns
Presolve time: 0.00s
Presolved: 392 rows, 317 columns, 1815 nonzeros
Variable types: 19 continuous, 298 integer (298 binary)
Found heuristic solution: objective 3.133356e+09

Root relaxation: objective 8.000024e+08, 72 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 8.0000e+08    0   72 3.1334e+09 8.0000e+08  74.5%     -    0s
H    0     0                     2.400019e+09 8.0000e+08  66.7%     -    0s
H    0     0                     2.220019e+09 8.0000e+08  64.0%     -    0s
     0     0 8.0000e+08    0   72 2.2200e+09 8.0000e+08  64.0%     -    0s
H    0     0                     2.200019e+09 8.0000e+08  63.6%     -    0s
     0     0 8.0000e+08    0   81 2.2000e+09 8.0000e+08  63.6%     -    0s
     0     0 8.0000e+08    0   77 2.2000e+09 8.0000e+08  63.6%     -    0s
     0     2 8.0000e+08    0   77 2.2000e+09 8.0000e+08  63.6%     -    0s
H  307   609                     2.066686e+09 8.0000e+08  61.3%   5.8    0s
H 1126   885                     1.950016e+09 8.0000e+08  59.0%   6.0    0s
H 1317   983                     1.900015e+09 8.0000e+08  57.9%   5.6    0s
H 1817  1173                     1.900015e+09 8.0000e+08  57.9%   5.0    0s
H 2656  1796                     1.900015e+09 8.0000e+08  57.9%   4.8    0s
H 8305  6287                     1.900015e+09 8.0000e+08  57.9%   3.4    0s
*10878  6870              99     1.808351e+09 8.0000e+08  55.8%   3.3    0s
*12677  7866              62     1.800016e+09 8.0000e+08  55.6%   3.3    1s
*17157 10811             118     1.800015e+09 8.0000e+08  55.6%   3.2    1s
H19145 11166                     1.750016e+09 8.0000e+08  54.3%   3.2    1s
H24736 14317                     1.700015e+09 8.0000e+08  52.9%   3.2    1s
H24874 14315                     1.700015e+09 8.0000e+08  52.9%   3.2    1s
H32097 17197                     1.633347e+09 8.0665e+08  50.6%   3.2    2s
H32123 16354                     1.600013e+09 8.1873e+08  48.8%   3.2    3s
 32158 16378 1.6000e+09  118  110 1.6000e+09 8.4564e+08  47.1%   3.2    5s
H32215 15596                     1.533346e+09 8.6063e+08  43.9%   3.3    6s
H32284 14860                     1.500013e+09 8.8136e+08  41.2%   3.4    9s
 32294 14867 1.2500e+09   46   93 1.5000e+09 8.8136e+08  41.2%   3.4   10s
 32446 14975 1.2500e+09   52   95 1.5000e+09 9.0001e+08  40.0%   3.5   15s

Interrupt request received

Cutting planes:
```

```
  Gomory: 8
  Implied bound: 13
  Projected implied bound: 1
  MIR: 19
  Flow cover: 17
  Zero half: 1
  RLT: 4
  Relax-and-lift: 17

Explored 57196 nodes (301282 simplex iterations) in 19.00 seconds
Thread count was 8 (of 8 available processors)

Solution count 10: 1.50001e+09 1.53335e+09 1.60001e+09 ... 1.80835e+09

Solve interrupted
Best objective 1.500012666667e+09, best bound 1.000006945369e+09, gap 33.3334%
```

It quickly becomes apparent that this model is quite a bit more difficult than the earlier `coins` model. The optimal solution is actually `1,200,000,000`, but finding that solution takes a while. After letting the model run for 10 seconds, we interrupt the run (by hitting CTRL-C, which produces the *Interrupt request received* message) and consider our options. Typing `m.optimize()` would resume the run from the point at which it was interrupted.

Changing parameters

Rather than continuing optimization on a difficult model like `glass4`, it is sometimes useful to try different parameter settings. When the lower bound moves slowly, as it does on this model, one potentially useful parameter is `MIPFocus`, which adjusts the high-level MIP solution strategy. Let us now set this parameter to value 1, which changes the focus of the MIP search to finding good feasible solutions. There are two ways to change the parameter value. You can either use method *m.setParam()*:

```
gurobi> m.setParam('MIPFocus', 1)
Changed value of parameter MIPFocus to 1
   Prev: 0   Min: 0   Max: 3   Default: 0
```

...or you can use the `m.Params` class...

```
gurobi> m.Params.MIPFocus = 1
Changed value of parameter MIPFocus to 1
   Prev: 0   Min: 0   Max: 3   Default: 0
```

Once the parameter has been changed, we call *m.reset()* to reset the optimization on our model and then *m.optimize()* to start a new optimization run:

```
gurobi> m.reset()
Discarded solution information
gurobi> m.optimize()
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)

Optimize a model with 396 rows, 322 columns and 1815 nonzeros
Model fingerprint: 0x541d0ad3
Variable types: 20 continuous, 302 integer (0 binary)
```

```
Coefficient statistics:
  Matrix range     [1e+00, 8e+06]
  Objective range  [1e+00, 1e+06]
  Bounds range     [1e+00, 8e+02]
  RHS range        [1e+00, 8e+06]
Presolve removed 4 rows and 5 columns
Presolve time: 0.00s
Presolved: 392 rows, 317 columns, 1815 nonzeros
Variable types: 19 continuous, 298 integer (298 binary)
Found heuristic solution: objective 3.133356e+09

Root relaxation: objective 8.000024e+08, 72 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 8.0000e+08    0   72 3.1334e+09 8.0000e+08  74.5%     -    0s
H    0     0                     2.400019e+09 8.0000e+08  66.7%     -    0s
H    0     0                     2.220019e+09 8.0000e+08  64.0%     -    0s
     0     0 8.0000e+08    0   72 2.2200e+09 8.0000e+08  64.0%     -    0s
H    0     0                     2.166685e+09 8.0000e+08  63.1%     -    0s
     0     0 8.0000e+08    0   72 2.1667e+09 8.0000e+08  63.1%     -    0s
     0     0 8.0000e+08    0   77 2.1667e+09 8.0000e+08  63.1%     -    0s
H    0     0                     2.133351e+09 8.0000e+08  62.5%     -    0s
     0     0 8.0000e+08    0   80 2.1334e+09 8.0000e+08  62.5%     -    0s
     0     0 8.0000e+08    0   80 2.1334e+09 8.0000e+08  62.5%     -    0s
     0     0 8.0000e+08    0   83 2.1334e+09 8.0000e+08  62.5%     -    0s
     0     0 8.0000e+08    0   78 2.1334e+09 8.0000e+08  62.5%     -    0s
     0     0 8.0000e+08    0   83 2.1334e+09 8.0000e+08  62.5%     -    0s
     0     0 8.0000e+08    0   83 2.1334e+09 8.0000e+08  62.5%     -    0s
     0     0 8.0000e+08    0   88 2.1334e+09 8.0000e+08  62.5%     -    0s
     0     0 8.0000e+08    0   66 2.1334e+09 8.0000e+08  62.5%     -    0s
H    0     0                     2.050017e+09 8.0000e+08  61.0%     -    0s
     0     2 8.0000e+08    0   65 2.0500e+09 8.0000e+08  61.0%     -    0s
H    1     4                     2.050017e+09 8.0000e+08  61.0%  74.0    0s
H    6     8                     2.000016e+09 8.0000e+08  60.0%  41.8    0s
H  130   128                     1.700015e+09 8.0000e+08  52.9%  12.7    0s
H  199   203                     1.644459e+09 8.0000e+08  51.4%  10.8    0s
H  213   213                     1.644459e+09 8.0000e+08  51.4%  10.8    1s
H  244   269                     1.633347e+09 8.0001e+08  51.0%  11.0    1s
  1428  1027 1.5333e+09   40   44 1.6333e+09 8.0001e+08  51.0%  15.5    5s
  3138  1602 1.3750e+09   58   22 1.6333e+09 8.0001e+08  51.0%  20.5   10s
* 4233  2185              66    1.600017e+09 8.0001e+08  50.0%  21.5   12s
* 4238  2082              67    1.550017e+09 8.0001e+08  48.4%  21.5   12s
H 4308  2026                     1.500016e+09 8.0001e+08  46.7%  21.6   14s
  4457  2226 1.1000e+09   36   65 1.5000e+09 8.0001e+08  46.7%  22.6   15s
H 4809  2136                     1.450016e+09 8.0001e+08  44.8%  23.4   16s
H 4908  2043                     1.400013e+09 8.0001e+08  42.9%  23.9   17s
H 5098  2027                     1.350013e+09 8.0001e+08  40.7%  24.8   18s
H 5282  1752                     1.200013e+09 8.0001e+08  33.3%  25.7   18s

Interrupt request received

Cutting planes:
  Gomory: 37
  Cover: 9
```

```
  Implied bound: 41
  MIR: 51
  Flow cover: 266
  RLT: 107
  Relax-and-lift: 99

Explored 5332 nodes (140122 simplex iterations) in 19.00 seconds
Thread count was 8 (of 8 available processors)

Solution count 10: 1.20001e+09 1.35001e+09 1.40001e+09 ... 1.64446e+09

Solve interrupted
Best objective 1.200012600000e+09, best bound 8.000066838804e+08, gap 33.3335%
```

Results are consistent with our expectations. We find a better solution sooner by shifting the focus towards finding feasible solutions (objective value `1.2e9` versus `1.5e9`).

The *setParam()* method is designed to be quite flexible and forgiving. It accepts wildcards as arguments, and it ignores character case. Thus, the following commands are all equivalent:

```
gurobi> m.setParam('NODELIMIT', 100)
gurobi> m.setParam('NodeLimit', 100)
gurobi> m.setParam('Node*', 100)
gurobi> m.setParam('N???Limit, 100)
```

You can use wildcards to get a list of matching parameters:

```
gurobi> m.setParam('*Cuts', 2)
Matching parameters: ['Cuts', 'CliqueCuts', 'CoverCuts', 'FlowCoverCuts',
'FlowPathCuts', 'GUBCoverCuts', 'ImpliedCuts', 'MIPSepCuts', 'MIRCuts', 'ModKCuts',
'NetworkCuts', 'SubMIPCuts', 'ZeroHalfCuts']
```

Note that `Model.Params` is a bit less forgiving than *setParam()*. In particular, wildcards are not allowed with this approach. You don't have to worry about capitalization of parameter names in either approach, though, so `m.Params.Heuristics` and `m.Params.heuristics` are equivalent.

The full set of available parameters can be browsed using the *paramHelp()* command. You can obtain further information on a specific parameter (e.g., `MIPGap`) by typing `paramHelp('MIPGap')`.

Parameter tuning tool

When confronted with the task of choosing parameter values that might lead to better performance on a model, the long list of Gurobi parameters may seem intimidating. To simplify the process, we include a simple automated parameter tuning tool. From the interactive shell, the command is `tune`:

```
gurobi> m = read('misc07')
gurobi> m.tune()
```

The tool tries a number of different parameter settings, and eventually outputs the best ones that it finds. For example:

```
Tested 29 parameter sets in 99.33s

Baseline parameter set: mean runtime 1.48s

Improved parameter set 1 (mean runtime 1.13s):

        FlowCoverCuts 1
        Aggregate 0

Improved parameter set 2 (mean runtime 1.22s):

        MIPFocus 1
```

In this case, it found that setting the `MIPFocus` parameter to 1 for model `misc07` reduced the runtime from 1.48 to 1.22.

Note that tuning is meant to give general suggestions for parameters that might help performance. You should make sure that the results it gives on one model are helpful on the full range of models you plan to solve. You may sometimes find that performance problems can't be fixed with parameter changes alone, particularly if your model has severe numerical issues.

Tuning is also available as a standalone program. From a command prompt, you can type:

```
> grbtune c:\gurobi903\win64\examples\data\p0033
```

Please consult the *Automated Tuning Tool* section of the Gurobi Reference Manual for more information.

Using a gurobi.env file

When you want to change the values of Gurobi parameters, you actually have several options available for doing so. We've already discussed parameter changes through the command-line tool (e.g., `gurobi_cl Threads=1 coins.lp`), and through interactive shell commands (e.g., `m.setParam('Threads', 1)`). Each of our language APIs also provides methods for setting parameters. The other option we'd like to mention now is the `gurobi.env` file.

Whenever the Gurobi library starts, it will look for file `gurobi.env` in the current working directory, and will apply any parameter changes contained therein. This is true whether the Gurobi library is invoked from the command-line, from the interactive shell, or from any of the Gurobi APIs. Parameter settings are stored one per line in this file, with the parameter name first, followed by at least one space, followed by the desired value. Lines beginning with the `#` sign are comments and are ignored. To give an example, the following (Linux) commands:

```
> echo "Threads 1" > gurobi.env
> gurobi_cl coins.lp
Using license file c:\gurobi\gurobi.lic
Using gurobi.env file
Set parameter LogFile to value gurobi.log
Set parameter Threads to value 1


Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
```

```
Copyright (c) 2019, Gurobi Optimization, LLC

Read LP format model from file coins.lp
Reading time = 0.00 seconds
: 4 rows, 9 columns, 16 nonzeros
Optimize a model with 4 rows, 9 columns and 16 nonzeros
Model fingerprint: 0xa0c5449c
Variable types: 4 continuous, 5 integer (0 binary)
Coefficient statistics:
  Matrix range     [6e-02, 7e+00]
  Objective range  [1e-02, 1e+00]
  Bounds range     [5e+01, 1e+03]
  RHS range        [0e+00, 0e+00]
Found heuristic solution: objective -0.0000000
Presolve removed 1 rows and 5 columns
Presolve time: 0.00s
Presolved: 3 rows, 4 columns, 9 nonzeros
Variable types: 0 continuous, 4 integer (0 binary)

Root relaxation: objective 1.134615e+02, 2 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  113.46154    0    1   -0.00000  113.46154      -     -    0s
H    0     0                       113.4500000  113.46154  0.01%     -    0s
     0     0  113.46154    0    1  113.45000  113.46154  0.01%     -    0s

Explored 1 nodes (2 simplex iterations) in 0.00 seconds
Thread count was 1 (of 8 available processors)

Solution count 2: 113.45 -0

Optimal solution found (tolerance 1.00e-04)
Best objective 1.134500000000e+02, best bound 1.134500000000e+02, gap 0.0000%
```

would read the new value of the `Threads` parameter from file `gurobi.env` and then optimize model `coins.lp` using one thread. Note that if the same parameter is changed in both *gurobi.env* and in your program (or through the Gurobi command-line), the value from `gurobi.env` will be overridden.

The distribution includes a sample `gurobi.env` file (in the `bin` directory). The sample includes every parameter, with the default value for each, but with all settings commented out.

Working with multiple models

The Gurobi shell allows you to work with multiple models simultaneously. For example...

```
gurobi> a = read('c:/gurobi903/win64/examples/data/p0033')
Read MPS format model from file c:/gurobi903/win64/examples/data/p0033.mps
Reading time = 0.00 seconds
```

```
P0033:  16 Rows, 33 Columns, 98 NonZeros.
gurobi> b = read('c:/gurobi903/win64/examples/data/stein9')
Read MPS format model from file c:/gurobi903/win64/examples/data/stein9.mps
Reading time = 0.00 seconds
STEIN9:  13 Rows, 9 Columns, 45 NonZeros.
```

The *models()* command gives a list of all active models.

```
gurobi> models()
Currently loaded models:
a : <gurobi.Model MIP instance P0033: 16 constrs, 33 vars, Parameter changes: LogFile=gurobi.log>
b : <gurobi.Model MIP instance STEIN9: 13 constrs, 9 vars, Parameter changes: LogFile=gurobi.log>
```

Note that parameters can be set for a particular model with the *Model.setParam()* method or the `Model.Params` class, or they can be changed for all models in the Gurobi shell by using the global *setParam()* method.

Help

The interactive shell includes an extensive help facility. To access it, simply type *help()* at the prompt. As previously mentioned, help is available for all of the important objects in the interface. For example, as explained in the help facility, you can type `help(Model)`, `help(Var)`, or `help(Constr)`. You can also obtain detailed help on any of the available methods on these objects. For example, `help(Model.setParam)` gives help on setting model parameters. You can also use a variable, or a method on a variable, to ask for help. For example, if variable `m` contains a Model object, then `help(m)` is equivalent to `help(Model)`, and `help(m.setParam)` is equivalent to `help(Model.setParam)`.

Interface customization

The Gurobi interactive shell lives within a full-featured scripting language. This allows you to perform a wide range of customizations to suit your particular needs. Creating custom functions requires some knowledge of the Python language, but you can achieve a lot by using a very limited set of language features.

Let us consider a simple example. Imagine that you store your models in a certain directory on your disk. Rather than having to type the full path whenever you read a model, you can create your own custom `read` method:

```
gurobi> def myread(filename):
.......   return read('/home/john/models/'+filename)
```

Note that the indentation of the second line is required.

Defining this function allows you to do the following:

```
gurobi> m = myread('stein9')
Read MPS format model from file /home/john/models/stein9.mps
```

If you don't want to type this function in each time you start the Gurobi shell, you can store it in a file. The file would look like the following:

```
from gurobipy import *

def myread(filename):
  return read('/home/john/models/'+filename)
```

The `from gurobipy import *` line is required in order to allow you to use the `read` method from the Gurobi shell in your custom function. The name of your customization file must end with a `.py` suffix. If the file is named `custom.py`, you would then type the following to import this function:

```
gurobi> from custom import *
```

One file can contain as many custom functions as you'd like (see `custom.py` in `<installdir>/examples/python` for an example). If you wish to make site-wide customizations, you can also customize the `gurobi.py` file that is included in `<installdir>/lib`.

Customization through callbacks

Another type of customization we'd like to touch on briefly can be achieved through Gurobi callbacks. Callbacks allow you to track the progress of the optimization process. For the sake of our example, let's say you want the MIP optimizer to run for 10 seconds before quitting, but you don't want it to terminate before it finds a feasible solution. The following callback method would implement this condition:

```
from gurobipy import *

def mycallback(model, where):
  if where == GRB.Callback.MIP:
    time = model.cbGet(GRB.Callback.RUNTIME)
    best = model.cbGet(GRB.Callback.MIP_OBJBST)
    if time > 10 and best < GRB.INFINITY:
      model.terminate()
```

Once you import this function (`from custom import *`), you can then say `m.optimize(mycallback)` to obtain the desired termination behavior. Alternatively, you could define your own custom optimize method that always invokes the callback:

```
def myopt(model):
  model.optimize(mycallback)
```

This would allow you to say `myopt(m)`.

You can pass arbitrary data into your callback through the model object. For example, if you set `m._mydata = 1` before calling *optimize()*, you can query `m._mydata` inside your callback function. Note that the names of user data fields must begin with an underscore.

This callback example is included in `<installdir>/examples/python/custom.py`.
Type `from custom import *` to import the callback and the *myopt()* function.

You can type `help(GRB.Callback)` for more information on callbacks. You can also refer to the `Callback` class documentation in the Gurobi Reference Manual.

The Gurobi Python Interface for Python Users

While the Gurobi installation includes everything you need to use Gurobi from within Python, we understand that some users would prefer to use Gurobi from within their own Python environment. Doing so requires you to install the `gurobipy` module. The steps for doing this depend on your platform. On Windows, you can double-click on the `pysetup` program in the Gurobi `<installdir>/bin` directory. This program will prompt you for the location of your Python installation; it handles all of the details of the installation.

Note that Gurobi distributes an Anaconda package, so you don't need to perform these steps if you are using Anaconda Python. Please refer to the instructions for setting up Gurobi for use within the Anaconda distribution for details.

Note that for this installation to succeed, your Python environment must be compatible with the Gurobi Python module. You should only install Gurobi libraries into a 64-bit Python shell. In addition, your Python version must be compatible. With this release, `gurobipy` can be used with Python 2.7, 3.6, or 3.7 on Windows.

As mentioned in the previous section, most of the information associated with a Gurobi model is stored in a set of *attributes*. Some attributes are associated with the variables of the model, some with the constraints of the model, and some with the model itself. After you optimize a model, for example, the solution is stored in the `X` variable attribute. Attributes that are computed by the Gurobi optimizer (such as the solution attribute) cannot be modified directly by the user, while those that represent input data (such as the `LB` attribute which stores variable lower bounds) can.

Each of the Gurobi language interfaces contains routines for querying or modifying attribute values. To retrieve or modify the value of a particular attribute, you simply pass the name of the attribute to the appropriate query or modification routine. In the C interface, for example, you'd make the following call to query the current solution value on variable 1:

```
double x1;
error = GRBgetdblattrelement(model, GRB_DBL_ATTR_X, 1, &x1);
```

This routine returns a single element from an array-valued attribute containing double-precision data. Routines are provided to query and modify scalar-valued and array-valued attributes of type `int`, `double`, `char`, or `char *`.

In the object oriented interfaces, you query or modify attribute values through the appropriate objects. For example, if variable `v` is a Gurobi variable object (a `GRBVar`), then the following calls would be used to modify the lower bound on `v`:

```
C++:    v.set(GRB_DoubleAttr_LB, 0.0)
Java:   v.set(GRB.DoubleAttr.LB, 0.0)
C#:     v.Set(GRB.DoubleAttr.LB, 0.0) or v.LB = 0.0
Python: v.lb = 0.0
```

The exact syntax for querying or modifying an attribute varies slightly from one language to another, but the basic approach remains consistent: you call the appropriate query or modification method using the name of the desired attribute as an argument.

The full list of Gurobi attributes can be found in the *Attributes* section of the Gurobi Reference Manual.

This section will work through a simple C example in order to illustrate the use of the Gurobi C interface. The example builds a simple Mixed Integer Programming model, optimizes it, and outputs the optimal objective value. This section assumes that you are already familiar with the C programming language. If not, a variety of books are available for learning the language (for example, *The C Programming Language*, by Kernighan and Ritchie).

Our example optimizes the following model:

$$\begin{array}{lrcrcrcl} \textbf{maximize} & x & + & y & + & 2\,z & & \\ \textbf{subject to} & x & + & 2\,y & + & 3\,z & \leq & 4 \\ & x & + & y & & & \geq & 1 \\ & & & x,\, y,\, z \text{ binary} & & & & \end{array}$$

Example mip1_c.c

This is the complete source code for our example (also available as `<installdir>/examples/c/mip1_c.c`)...

```
/* Copyright 2020, Gurobi Optimization, LLC */

/* This example formulates and solves the following simple MIP model:

     maximize    x +   y + 2 z
     subject to  x + 2 y + 3 z <= 4
                 x +   y       >= 1
                 x, y, z binary
*/

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

int
main(int   argc,
     char *argv[])
{
  GRBenv   *env   = NULL;
  GRBmodel *model = NULL;
  int       error = 0;
  double    sol[3];
  int       ind[3];
  double    val[3];
  double    obj[3];
  char      vtype[3];
  int       optimstatus;
```

```
double    objval;

/* Create environment */
error = GRBemptyenv(&env);
if (error) goto QUIT;

error = GRBsetstrparam(env, "LogFile", "mip1.log");
if (error) goto QUIT;

error = GRBstartenv(env);
if (error) goto QUIT;

/* Create an empty model */
error = GRBnewmodel(env, &model, "mip1", 0, NULL, NULL, NULL, NULL, NULL);
if (error) goto QUIT;

/* Add variables */
obj[0] = 1; obj[1] = 1; obj[2] = 2;
vtype[0] = GRB_BINARY; vtype[1] = GRB_BINARY; vtype[2] = GRB_BINARY;
error = GRBaddvars(model, 3, 0, NULL, NULL, NULL, obj, NULL, NULL, vtype,
                   NULL);
if (error) goto QUIT;

/* Change objective sense to maximization */
error = GRBsetintattr(model, GRB_INT_ATTR_MODELSENSE, GRB_MAXIMIZE);
if (error) goto QUIT;

/* First constraint: x + 2 y + 3 z <= 4 */
ind[0] = 0; ind[1] = 1; ind[2] = 2;
val[0] = 1; val[1] = 2; val[2] = 3;

error = GRBaddconstr(model, 3, ind, val, GRB_LESS_EQUAL, 4.0, "c0");
if (error) goto QUIT;

/* Second constraint: x + y >= 1 */
ind[0] = 0; ind[1] = 1;
val[0] = 1; val[1] = 1;

error = GRBaddconstr(model, 2, ind, val, GRB_GREATER_EQUAL, 1.0, "c1");
if (error) goto QUIT;

/* Optimize model */
error = GRBoptimize(model);
if (error) goto QUIT;

/* Write model to 'mip1.lp' */
error = GRBwrite(model, "mip1.lp");
if (error) goto QUIT;

/* Capture solution information */
error = GRBgetintattr(model, GRB_INT_ATTR_STATUS, &optimstatus);
if (error) goto QUIT;

error = GRBgetdblattr(model, GRB_DBL_ATTR_OBJVAL, &objval);
if (error) goto QUIT;
```

```
  error = GRBgetdblattrarray(model, GRB_DBL_ATTR_X, 0, 3, sol);
  if (error) goto QUIT;

  printf("\nOptimization complete\n");
  if (optimstatus == GRB_OPTIMAL) {
    printf("Optimal objective: %.4e\n", objval);

    printf("  x=%.0f, y=%.0f, z=%.0f\n", sol[0], sol[1], sol[2]);
  } else if (optimstatus == GRB_INF_OR_UNBD) {
    printf("Model is infeasible or unbounded\n");
  } else {
    printf("Optimization was stopped early\n");
  }

QUIT:

  /* Error reporting */
  if (error) {
    printf("ERROR: %s\n", GRBgeterrormsg(env));
    exit(1);
  }

  /* Free model */
  GRBfreemodel(model);

  /* Free environment */
  GRBfreeenv(env);

  return 0;
}
```

Example details

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

The example begins by including a few include files. Gurobi C applications should always start by including `gurobi_c.h`, along with the standard C include files (`stdlib.h` and `stdio.h`).

Creating the environment

After declaring the necessary program variables, the example continues by creating an environment, by first requesting an empty environment, then setting some options -- as log file name -- and then starting the environment.

```
  /* Create environment */
  error = GRBemptyenv(&env);
  if (error) goto QUIT;

  error = GRBsetstrparam(env, "LogFile", "mip1.log");
  if (error) goto QUIT;
```

```
error = GRBstartenv(env);
if (error) goto QUIT;
```

Later requests to create optimization models will always require an active environment, so environment creation should always be the first step when using the Gurobi optimizer.

Note that environment creation may fail, so you should check the return value of the call.

Creating the model

Once an environment has been created, the next step is to create a model. A Gurobi model holds a single optimization problem. It consists of a set of variables, a set of constraints, and the associated attributes (variable bounds, objective coefficients, variable integrality types, constraint senses, constraint right-hand side values, etc.). The first step towards building a model that contains all of this information is to create an empty model object:

```
/* Create an empty model */
error = GRBnewmodel(env, &model, "mip1", 0, NULL, NULL, NULL, NULL, NULL);
if (error) goto QUIT;
```

The first argument to *GRBnewmodel()* is the previously created environment. The second is a pointer to the location where the pointer to the new model should be stored. The third is the name of the model. The fourth is the number of variables to initially add to the model. Since we're creating an empty model, the number of initial variables is 0. The remaining arguments would describe the initial variables (lower bounds, upper bounds, variable types, etc.), had they been present.

Adding variables to the model

Once we create a Gurobi model, we can start adding variables and constraints to it. In our example, we'll begin by adding variables:

```
/* Add variables */
obj[0] = 1; obj[1] = 1; obj[2] = 2;
vtype[0] = GRB_BINARY; vtype[1] = GRB_BINARY; vtype[2] = GRB_BINARY;
error = GRBaddvars(model, 3, 0, NULL, NULL, NULL, obj, NULL, NULL, vtype,
                   NULL);
if (error) goto QUIT;
```

The first argument to *GRBaddvars()* is the model to which the variables are being added. The second is the number of added variables (3 in our example).

Arguments three through six describe the constraint matrix coefficients associated with the new variables. The third argument gives the number of non-zero constraint matrix entries associated with the new variables, and the next three arguments give details on these non-zeros. In our example, we'll be adding these non-zeros when we add the constraints. Thus, the non-zero count here is zero, and the following three arguments are all `NULL`.

The seventh argument to *GRBaddvars()* is the linear objective coefficient for each new variable. Since our example aims to maximize the objective, and by default Gurobi will minimize the objective, we'll need to change the objective sense. This is done in the next statement. Note we could have multiplied the objective coefficients by -1 instead (since maximizing $c'x$ is equivalent to minimizing $-c'x$).

The next two arguments specify the lower and upper bounds of the variables, respectively. The `NULL` values indicate that these variables should take their default values (0.0 and 1.0 for binary variables).

The tenth argument specifies the types of the variables. In this example, the variables are all binary (`GRB_BINARY`).

The final argument gives the names of the variables. In this case, we allow the variable names to take their default values (`x0, x1, and x2`).

Changing the objective sense

As we just noted, the default sense for the objective function is minimization. Since our example aims to maximize the objective, we need to modify the `ModelSense` attribute:

```
/* Change objective sense to maximization */
error = GRBsetintattr(model, GRB_INT_ATTR_MODELSENSE, GRB_MAXIMIZE);
if (error) goto QUIT;
```

Adding constraints to the model

Once the new variables are integrated into the model, the next step is to add our two linear constraints. These constraints are added through the *GRBaddconstr()* routine. To add a constraint, you must specify several pieces of information, including the non-zero values associated with the constraint, the constraint sense, the right-hand side value, and the constraint name. These are all specified as arguments to *GRBaddconstr()*:

```
/* First constraint: x + 2 y + 3 z <= 4 */
ind[0] = 0; ind[1] = 1; ind[2] = 2;
val[0] = 1; val[1] = 2; val[2] = 3;

error = GRBaddconstr(model, 3, ind, val, GRB_LESS_EQUAL, 4.0, "c0");
if (error) goto QUIT;
```

The first argument of *GRBaddconstr()* is the model to which the constraint is being added. The second is the total number of non-zero coefficients associated with the new constraint. The next two arguments describe the non-zeros in the new constraint. Constraint coefficients are specified using a list of index-value pairs, one for each non-zero value. In our example, the first constraint to be added is $x+2y+3z \leq 4$. We have chosen to make x the first variable in our constraint matrix, y the second, and z the third (note that this choice is arbitrary). Given our variable ordering choice,

the index-value pairs that are required for our first constraint are (0, 1.0), (1, 2.0), and (2, 3.0). These pairs are placed in the `ind` and `val` arrays.

The fifth argument to *GRBaddconstr()* provides the sense of the new constraint. Possible values are `GRB_LESS_EQUAL`, `GRB_GREATER_EQUAL`, or `GRB_EQUAL`. The sixth argument gives the right-hand side value. The final argument gives the name of the constraint (we allow the constraint to take its default name here by specifying `NULL` for the argument). The second constraint is added in a similar fashion.

Note that routine *GRBaddconstrs()* would allow you to add both constraints in a single call. The arguments for this routine are much more complex, though, without providing any significant advantages, so we recommend that you add one constraint at a time.

Optimizing the model

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

```
/* Optimize model */
error = GRBoptimize(model);
if (error) goto QUIT;
```

This routine performs the optimization and populates several internal model attributes, including the status of the optimization, the solution, etc. Once the function returns, we can query the values of these attributes. In particular, we can query the status of the optimization process by retrieving the value of the `Status` attribute...

```
/* Capture solution information */
error = GRBgetintattr(model, GRB_INT_ATTR_STATUS, &optimstatus);
if (error) goto QUIT;
```

The optimization status has many possible values. An optimal solution to the model may have been found, or the model may have been determined to be infeasible or unbounded, or the solution process may have been interrupted. A list of possible statuses can be found in the Gurobi Reference Manual. For our example, we know that the model is feasible, and we haven't modified any parameters that might cause the optimization to stop early (e.g., a time limit), so the status will be `GRB_OPTIMAL`.

Another important model attribute is the value of the objective function for the computed solution. This is accessed through this call:

```
error = GRBgetdblattr(model, GRB_DBL_ATTR_OBJVAL, &objval);
if (error) goto QUIT;
```

Note that this call would return a non-zero error result if no solution was found for this model.

Once we know that the model was solved, we can extract the `X` attribute of the model, which contains the value for each variable in the computed solution:

```
error = GRBgetdblattrarray(model, GRB_DBL_ATTR_X, 0, 3, sol);
if (error) goto QUIT;
```

This routine retrieves the values of an array-valued attribute. The third and fourth arguments indicate the index of the first array element to be retrieved, and the number of elements to retrieve, respectively. In this example we retrieve entries 0 through 2 (i.e., all three of them)

Error reporting

We would like to point out one additional aspect of the example. Almost all of the Gurobi methods return an error code. The code will typically be zero, indicating that no error was encountered, but it is important to check the value of the code in case an error arises.

While you may want to print a specialized error code at each point where an error may occur, the Gurobi interface provides a more flexible facility for reporting errors. The *GRBgeterrormsg()* routine returns a textual description of the most recent error associated with an environment:

```
/* Error reporting */
if (error) {
  printf("ERROR: %s\n", GRBgeterrormsg(env));
  exit(1);
}
```

Once the error reporting is done, the only remaining task in our example is to release the resources associated with our optimization task. In this case, we populated one model and created one environment. We call `GRBfreemodel(model)` to free the model, and `GRBfreeenv(env)` to free the environment (in that order).

Building and running the example

To build and run the example, please refer to the files in `<installdir>/examples/build`. For Windows platforms, this directory contains `C_examples_2015.sln` and `C_examples_2017.sln` (Visual Studio 2015 and 2017 solution files for the C examples). Double-clicking on the solution file will bring up Visual Studio. Clicking on the `mip1_c` project, and then selecting *Run* from the *Build* menu will compile and run the example.

The C example directory `<installdir>/examples/c` contains a number of examples. We encourage you to browse and modify them in order to become more familiar with the Gurobi C interface. We also encourage you to read the Gurobi Example Tour for more information.

This section will work through a simple C++ example in order to illustrate the use of the Gurobi C++ interface. The example builds a model, optimizes it, and outputs the optimal objective value. This section assumes that you are already familiar with the C++ programming language. If not, a variety of books are available for learning the language (for example, *The C++ Programming Language*, by Stroustrup).

Our example optimizes the following model:

$$\begin{array}{lrcrcrcl} \textbf{maximize} & x & + & y & + & 2\,z & & \\ \textbf{subject to} & x & + & 2\,y & + & 3\,z & \leq & 4 \\ & x & + & y & & & \geq & 1 \\ & & & \multicolumn{5}{c}{x, y, z \text{ binary}} \end{array}$$

Note that this is the same model that was modeled and optimized in the C Interface section.

Example mip1_c++.cpp

This is the complete source code for our example (also available in `<installdir>/examples/c++/mip1_c++.cpp`)...

```
/* Copyright 2020, Gurobi Optimization, LLC */

/* This example formulates and solves the following simple MIP model:

     maximize    x +   y + 2 z
     subject to  x + 2 y + 3 z <= 4
                 x +   y       >= 1
                 x, y, z binary
*/

#include "gurobi_c++.h"
using namespace std;

int
main(int   argc,
     char *argv[])
{
  try {

    // Create an environment
    GRBEnv env = GRBEnv(true);
    env.set("LogFile", "mip1.log");
    env.start();
```

```
    // Create an empty model
    GRBModel model = GRBModel(env);

    // Create variables
    GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB_BINARY, "x");
    GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB_BINARY, "y");
    GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB_BINARY, "z");

    // Set objective: maximize x + y + 2 z
    model.setObjective(x + y + 2 * z, GRB_MAXIMIZE);

    // Add constraint: x + 2 y + 3 z <= 4
    model.addConstr(x + 2 * y + 3 * z <= 4, "c0");

    // Add constraint: x + y >= 1
    model.addConstr(x + y >= 1, "c1");

    // Optimize model
    model.optimize();

    cout << x.get(GRB_StringAttr_VarName) << " "
         << x.get(GRB_DoubleAttr_X) << endl;
    cout << y.get(GRB_StringAttr_VarName) << " "
         << y.get(GRB_DoubleAttr_X) << endl;
    cout << z.get(GRB_StringAttr_VarName) << " "
         << z.get(GRB_DoubleAttr_X) << endl;

    cout << "Obj: " << model.get(GRB_DoubleAttr_ObjVal) << endl;

  } catch(GRBException e) {
    cout << "Error code = " << e.getErrorCode() << endl;
    cout << e.getMessage() << endl;
  } catch(...) {
    cout << "Exception during optimization" << endl;
  }

  return 0;
}
```

Example details

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

The example begins by including file `gurobi_c++.h`. Gurobi C++ applications should always start by including this file.

Creating the environment

The first executable statement in our example obtains a Gurobi environment (using the `GRBEnv()` constructor):

```
// Create an environment
GRBEnv env = GRBEnv(true);
env.set("LogFile", "mip1.log");
env.start();
```

In this call we requested an *empty* environment, choose a log file, and started the environment.

Later calls to create an optimization model will always require an environment, so environment creation is typically the first step in a Gurobi application.

Creating the model

Once an environment has been created, the next step is to create a model. A Gurobi model holds a single optimization problem. It consists of a set of variables, a set of constraints, and the associated attributes (variable bounds, objective coefficients, variable integrality types, constraint senses, constraint right-hand side values, etc.). The first step towards building a model that contains all of this information is to create an empty model object:

```
// Create an empty model
GRBModel model = GRBModel(env);
```

The constructor takes the previously created environment as its argument.

Adding variables to the model

The next step in our example is to add variables to the model.

```
// Create variables
GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB_BINARY, "x");
GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB_BINARY, "y");
```

Variables are added through the `addVar()` method on the model object (or `addVars()` if you wish to add more than one at a time). A variable is always associated with a particular model.

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 binary 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 the Gurobi Reference Manual for further details.

Setting the objective

The next step in the example is to set the optimization objective:

```
// Set objective: maximize x + y + 2 z
model.setObjective(x + y + 2 * z, GRB_MAXIMIZE);
```

The objective is built here using overloaded operators. The C++ API overloads the arithmetic operators to allow you to build linear and quadratic expressions involving Gurobi variables.

The second argument indicates that the sense is maximization.

Note that while this simple example builds the objective in a single statement using an explicit list of terms, more complex programs will typically build it incrementally. For example:

```
GRBLinExpr obj = 0.0;
obj += x;
obj += y;
obj += 2*z;
model.setObjective(obj, GRB_MAXIMIZE);
```

Adding constraints to the model

The next step in the example is to add the linear constraints. The first constraint is added here:

```
// Add constraint: x + 2 y + 3 z <= 4
model.addConstr(x + 2 * y + 3 * z <= 4, "c0");
```

As with variables, constraints are always associated with a specific model. They are created using the *addConstr()* or *addConstrs()* methods on the model object.

We again use overloaded arithmetic operators to build the linear expression. The comparison operators are also overloaded to make it easier to build constraints.

The second argument to `addConstr` gives the (optional) constraint name.

Again, this simple example builds the linear expression for the constraint in a single statement using an explicit list of terms. More complex programs will typically build the expression incrementally.

The second constraint in our model is added with this similar call:

```
// Add constraint: x + y >= 1
model.addConstr(x + y >= 1, "c1");
```

Optimizing the model

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

```
// Optimize model
model.optimize();
```

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

Reporting results - attributes

Once the optimization is complete, we can query the values of the attributes. In particular, we can query the `VarName` and `X` attributes to obtain the name and solution value of each variable:

```
cout << x.get(GRB_StringAttr_VarName) << " "
     << x.get(GRB_DoubleAttr_X) << endl;
```

We can also query the `ObjVal` attribute on the model to obtain the objective value for the current solution:

```
cout << "Obj: " << model.get(GRB_DoubleAttr_ObjVal) << endl;
```

The names and types of all model, variable, and constraint attributes can be found in the `Attributes` section of the Gurobi Reference Manual.

Error handling

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

```
} catch(GRBException e) {
  cout << "Error code = " << e.getErrorCode() << endl;
  cout << e.getMessage() << endl;
} catch(...) {
  cout << "Exception during optimization" << endl;
}
```

Building and running the example

To build and run the example, we refer the user to the files in `<installdir>/examples/build`. For Windows platforms, this directory contains `C++_examples_2015.sln` and `C++_examples_2017.sln` (Visual Studio 2015 and 2017 solution files for the C++ examples). Double-clicking on the solution file will bring up Visual Studio. Clicking on the `mip1_c++` project, and then selecting *Run* from the *Build* menu will compile and run the example.

If you want to create your own project or makefile to build a C++ program that calls Gurobi, the details will depend on your platform and development environment, but we'd like to point out a few common pitfalls:

- Be sure to choose the Gurobi C++ library that is compatible with your Visual Studio version and your choice of runtime library (Gurobi supports runtime library options `/MD`, `/MDd`, `/MT`, and `/MTd`). To give an example, use file `gurobi_c++md2015.lib` when you choose runtime library option /MD in Visual Studio 2015. Similarly, use file `gurobi_c++mtd2017.lib` when you choose runtime library option /MTd in Visual Studio 2017.

- A C++ program that uses Gurobi must link in both the Gurobi C++ library `gurobi_c++mt2015.lib` *and* the Gurobi C library `gurobi90.lib`.

The C++ example directory `<installdir>/examples/c++` contains a number of examples. We encourage you to browse and modify them in order to become more familiar with the Gurobi C++ interface. We also encourage you to read the Gurobi Example Tour for more information.

This section will work through a simple Java example in order to illustrate the use of the Gurobi Java interface. The example builds a model, optimizes it, and outputs the optimal objective value. This section assumes that you are already familiar with the Java programming language. If not, a variety of books and websites are available for learning the language (for example, the online Java tutorials).

If you are using an integrated development environment like Eclipse®, we recommend that you import the file `<installdir>/lib/gurobi-javadoc.jar`, which contains Javadoc documentation for the Gurobi Java interface.

Our example optimizes the following model:

$$
\begin{array}{lrcrcrcl}
\textbf{maximize} & x & + & y & + & 2\,z & & \\
\textbf{subject to} & x & + & 2\,y & + & 3\,z & \leq & 4 \\
 & x & + & y & & & \geq & 1 \\
 & & & \multicolumn{5}{c}{x, y, z \text{ binary}}
\end{array}
$$

Note that this is the same model that was modeled and optimized in the C Interface section.

Example Mip1.java

This is the complete source code for our example (also available in `<installdir>/examples/java/Mip1.java`)...

```
/* Copyright 2020, Gurobi Optimization, LLC */

/* This example formulates and solves the following simple MIP model:

     maximize    x +   y + 2 z
     subject to  x + 2 y + 3 z <= 4
                 x +   y       >= 1
                 x, y, z binary
*/

import gurobi.*;

public class Mip1 {
  public static void main(String[] args) {
    try {

      // Create empty environment, set options, and start
      GRBEnv env = new GRBEnv(true);
      env.set("logFile", "mip1.log");
      env.start();
```

```
      // Create empty model
      GRBModel  model = new GRBModel(env);

      // Create variables
      GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "x");
      GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "y");
      GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "z");

      // Set objective: maximize x + y + 2 z
      GRBLinExpr expr = new GRBLinExpr();
      expr.addTerm(1.0, x); expr.addTerm(1.0, y); expr.addTerm(2.0, z);
      model.setObjective(expr, GRB.MAXIMIZE);

      // Add constraint: x + 2 y + 3 z <= 4
      expr = new GRBLinExpr();
      expr.addTerm(1.0, x); expr.addTerm(2.0, y); expr.addTerm(3.0, z);
      model.addConstr(expr, GRB.LESS_EQUAL, 4.0, "c0");

      // Add constraint: x + y >= 1
      expr = new GRBLinExpr();
      expr.addTerm(1.0, x); expr.addTerm(1.0, y);
      model.addConstr(expr, GRB.GREATER_EQUAL, 1.0, "c1");

      // Optimize model
      model.optimize();

      System.out.println(x.get(GRB.StringAttr.VarName)
                         + " " +x.get(GRB.DoubleAttr.X));
      System.out.println(y.get(GRB.StringAttr.VarName)
                         + " " +y.get(GRB.DoubleAttr.X));
      System.out.println(z.get(GRB.StringAttr.VarName)
                         + " " +z.get(GRB.DoubleAttr.X));

      System.out.println("Obj: " + model.get(GRB.DoubleAttr.ObjVal));

      // Dispose of model and environment
      model.dispose();
      env.dispose();

    } catch (GRBException e) {
      System.out.println("Error code: " + e.getErrorCode() + ". " +
                         e.getMessage());
    }
  }
}
```

Example details

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

The example begins by importing the Gurobi classes (`import gurobi.*`). Gurobi Java applications

should always start with this line.

Creating the environment

The first executable statement in our example obtains a Gurobi environment (using the `GRBEnv()` constructor):

```
// Create empty environment, set options, and start
GRBEnv env = new GRBEnv(true);
env.set("logFile", "mip1.log");
env.start();
```

In this call we requested an *empty* environment, choose a log file, and started the environment.

Later calls to create an optimization model will always require an environment, so environment creation is typically the first step in a Gurobi application. The constructor argument specifies the name of the log file.

Creating the model

Once an environment has been created, the next step is to create a model. A Gurobi model holds a single optimization problem. It consists of a set of variables, a set of constraints, and the associated attributes (variable bounds, objective coefficients, variable integrality types, constraint senses, constraint right-hand side values, etc.). The first step towards building a model that contains all of this information is to create an empty model object:

```
// Create empty model
GRBModel  model = new GRBModel(env);
```

The constructor takes the previously created environment as its argument.

Adding variables to the model

The next step in our example is to add variables to the model.

```
// Create variables
GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "x");
GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "y");
```

Variables are added through the `addVar()` method on a model object (or `addVars()` if you wish to add more than one at a time). A variable is always associated with a particular model.

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 binary 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 the Gurobi Reference Manual for further details.

Setting the objective

The next step in the example is to set the optimization objective:

```
// Set objective: maximize x + y + 2 z
GRBLinExpr expr = new GRBLinExpr();
expr.addTerm(1.0, x); expr.addTerm(1.0, y); expr.addTerm(2.0, z);
model.setObjective(expr, GRB.MAXIMIZE);
```

The objective must be a linear or quadratic function of the variables in the model. In our example, we build our objective by first constructing an empty linear expression and adding three terms to it.

The second argument to `setObjective` indicates that the optimization sense is maximization.

Adding constraints to the model

The next step in the example is to add the linear constraints. The first constraint is added here:

```
// Add constraint: x + 2 y + 3 z <= 4
expr = new GRBLinExpr();
expr.addTerm(1.0, x); expr.addTerm(2.0, y); expr.addTerm(3.0, z);
model.addConstr(expr, GRB.LESS_EQUAL, 4.0, "c0");
```

As with variables, constraints are always associated with a specific model. They are created using the *addConstr()* or *addConstrs()* methods on the model object.

The first argument to *addConstr()* is the left-hand side of the constraint. We built the left-hand side by first creating an empty linear expression object, and then adding three terms to it. The second argument is the constraint sense (`GRB_LESS_EQUAL`, `GRB_GREATER_EQUAL`, or `GRB_EQUAL`). The third argument is the right-hand side (a constant in our example). The final argument is the constraint name. Several signatures are available for *addConstr()*. Please consult the Gurobi Reference Manual for details.

The second constraint is created in a similar manner:

```
// Add constraint: x + y >= 1
expr = new GRBLinExpr();
expr.addTerm(1.0, x); expr.addTerm(1.0, y);
model.addConstr(expr, GRB.GREATER_EQUAL, 1.0, "c1");
```

Optimizing the model

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

```
// Optimize model
model.optimize();
```

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

Reporting results - attributes

Once the optimization is complete, we can query the values of the attributes. In particular, we can query the `VarName` and `X` attributes to obtain the name and solution value for each variable:

```
System.out.println(x.get(GRB.StringAttr.VarName)
                   + " " +x.get(GRB.DoubleAttr.X));
```

We can also query the `ObjVal` attribute on the model to obtain the objective value for the current solution:

```
System.out.println("Obj: " + model.get(GRB.DoubleAttr.ObjVal));
```

The names and types of all model, variable, and constraint attributes can be found in the `Attributes` section of the Gurobi Reference Manual.

Cleaning up

The example concludes with `dispose` calls:

```
// Dispose of model and environment
model.dispose();
```

These reclaim the resources associated with the model and environment. Garbage collection would reclaim these eventually, but if your program doesn't exit immediately after performing the optimization, it is best to reclaim them explicitly.

Note that all models associated with an environment must be disposed before the environment itself is disposed.

Error handling

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

```
} catch (GRBException e) {
  System.out.println("Error code: " + e.getErrorCode() + ". " +
                     e.getMessage());
}
```

Building and running the example

To build and run the example, please refer to the files in `<installdir>/examples/build`. For Windows platforms, this directory contains `runjava.bat`, a simple script to compile and run a java example. Say `runjava Mip1` to run this example.

The Java example directory `<installdir>/examples/java` contains a number of examples. We encourage you to browse and modify them in order to become more familiar with the Gurobi Java interface. We also encourage you to read the Gurobi Example Tour for more information.

.NET Interface (C#)

This section assumes that you are already familiar with the C# programming language. If not, a variety of books and websites are available for learning the language (for example, the Microsoft online C# documentation).

While C# programs have historically only run on Windows machines, Microsoft has recently released an open-source, cross-platform .NET implementation (.NET Core) that allows you to run C# programs on Linux and Mac OS as well. You will find the Gurobi library that supports .NET Core 2 in `<installdir>/lib/gurobi90.netstandard20.dll`. For more information on how to obtain and use .NET Core, please refer to the .NET Tutorial.

We will work through a simple C# example in order to illustrate the use of the Gurobi .NET interface. The example builds a model, optimizes it, and outputs the optimal objective value.

Our example optimizes the following model:

$$
\begin{array}{lrcrcrcl}
\textbf{maximize} & x & + & y & + & 2\,z & & \\
\textbf{subject to} & x & + & 2\,y & + & 3\,z & \leq & 4 \\
 & x & + & y & & & \geq & 1 \\
 & & & \multicolumn{5}{c}{x, y, z \text{ binary}}
\end{array}
$$

Note that this is the same model that was modeled and optimized in the C Interface section.

Example mip1_cs.cs

This is the complete source code for our example (also available in `<installdir>/examples/c#/mip1_cs.cs`)...

```
/* Copyright 2020, Gurobi Optimization, LLC */

/* This example formulates and solves the following simple MIP model:

     maximize    x +   y + 2 z
     subject to  x + 2 y + 3 z <= 4
                 x +   y       >= 1
                 x, y, z binary
*/

using System;
using Gurobi;

class mip1_cs
{
  static void Main()
  {
```

```
    try {

      // Create an empty environment, set options and start
      GRBEnv    env   = new GRBEnv(true);
      env.Set("LogFile", "mip1.log");
      env.Start();

      // Create empty model
      GRBModel  model = new GRBModel(env);

      // Create variables
      GRBVar x = model.AddVar(0.0, 1.0, 0.0, GRB.BINARY, "x");
      GRBVar y = model.AddVar(0.0, 1.0, 0.0, GRB.BINARY, "y");
      GRBVar z = model.AddVar(0.0, 1.0, 0.0, GRB.BINARY, "z");

      // Set objective: maximize x + y + 2 z
      model.SetObjective(x + y + 2 * z, GRB.MAXIMIZE);

      // Add constraint: x + 2 y + 3 z <= 4
      model.AddConstr(x + 2 * y + 3 * z <= 4.0, "c0");

      // Add constraint: x + y >= 1
      model.AddConstr(x + y >= 1.0, "c1");

      // Optimize model
      model.Optimize();

      Console.WriteLine(x.VarName + " " + x.X);
      Console.WriteLine(y.VarName + " " + y.X);
      Console.WriteLine(z.VarName + " " + z.X);

      Console.WriteLine("Obj: " + model.ObjVal);

      // Dispose of model and env
      model.Dispose();
      env.Dispose();

    } catch (GRBException e) {
      Console.WriteLine("Error code: " + e.ErrorCode + ". " + e.Message);
    }
  }
}
```

Example details

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

The example begins by importing the Gurobi namespace (`using Gurobi`). Gurobi .NET applications should always start with this line.

Creating the environment

The first executable statement in our example obtains a Gurobi environment (using the `GRBEnv()` constructor):

```
// Create an empty environment, set options and start
GRBEnv    env   = new GRBEnv(true);
env.Set("LogFile", "mip1.log");
env.Start();
```

In this case, we create an empty environment, select a log file to use, and start the environment for latter use.

Later calls to create an optimization model will always require an active environment, so environment creation is typically the first step in a Gurobi application. The constructor argument specifies the name of the log file.

Creating the model

Once an environment has been created, the next step is to create a model. A Gurobi model holds a single optimization problem. It consists of a set of variables, a set of constraints, and the associated attributes (variable bounds, objective coefficients, variable integrality types, constraint senses, constraint right-hand side values, etc.). The first step towards building a model that contains all of this information is to create an empty model object:

```
// Create empty model
GRBModel  model = new GRBModel(env);
```

The constructor takes the previously created environment as its argument.

Adding variables to the model

The next step in our example is to add variables to the model.

```
// Create variables
GRBVar x = model.AddVar(0.0, 1.0, 0.0, GRB.BINARY, "x");
GRBVar y = model.AddVar(0.0, 1.0, 0.0, GRB.BINARY, "y");
```

Variables are added through the `AddVar()` method on a model object (or `AddVars` if you wish to add more than one at a time). A variable is always associated with a particular model.

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 binary 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 the Gurobi Reference Manual for further details.

Setting the objective

The next step in the example is to set the optimization objective:

```
// Set objective: maximize x + y + 2 z
model.SetObjective(x + y + 2 * z, GRB.MAXIMIZE);
```

The objective is built here using overloaded operators. The C# API overloads the arithmetic operators to allow you to build linear and quadratic expressions involving Gurobi variables.

The second argument indicates that the sense is maximization.

Note that while this simple example builds the objective in a single statement using an explicit list of terms, more complex programs will typically build it incrementally. For example:

```
GRBLinExpr obj = 0.0;
obj.AddTerm(1.0, x);
obj.AddTerm(1.0, y);
obj.AddTerm(2.0, z);
model.SetObjective(obj, GRB.MAXIMIZE);
```

Adding constraints to the model

The next step in the example is to add the linear constraints:

```
// Add constraint: x + 2 y + 3 z <= 4
model.AddConstr(x + 2 * y + 3 * z <= 4.0, "c0");
```

As with variables, constraints are always associated with a specific model. They are created using the *AddConstr()* or *AddConstrs()* methods on the model object.

We again use overloaded arithmetic operators to build linear expressions. The comparison operators are also overloaded to make it easier to build constraints.

The second argument to `AddConstr` gives the constraint name.

The Gurobi .NET interface also allows you to add constraints by building linear expressions in a term-by-term fashion:

```
GRBLinExpr expr = 0.0;
expr.AddTerm(1.0, x);
expr.AddTerm(2.0, y);
expr.AddTerm(3.0, z);
model.AddConstr(expr, GRB.LESS_EQUAL, 4.0, "c0");
```

This particular *AddConstr()* signature takes a linear expression that captures the left-hand side of the constraint as its first argument, the sense of the constraint as its second argument, and a linear expression that captures the right-hand side of the constraint as its third argument. The constraint name is given as the fourth argument.

Optimizing the model

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

```
// Optimize model
model.Optimize();
```

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

Reporting results - attributes

Once the optimization is complete, we can query the values of the attributes (which are implemented as .NET *properties*). In particular, we can query the `VarName` and `X` attributes to obtain the name and solution value for each variable:

```
Console.WriteLine(x.VarName + " " + x.X);
Console.WriteLine(y.VarName + " " + y.X);
```

We can also query the `ObjVal` attribute on the model to obtain the objective value for the current solution:

```
Console.WriteLine("Obj: " + model.ObjVal);
```

The names and types of all model, variable, and constraint attributes can be found in the *Attributes* section of the Gurobi Reference Manual.

Cleaning up

The example concludes with `Dispose` calls:

```
// Dispose of model and env
model.Dispose();
```

These reclaim the resources associated with the model and environment. Garbage collection would reclaim these eventually, but if your program doesn't exit immediately after performing the optimization, it is best to reclaim them explicitly.

Note that all models associated with an environment must be disposed before the environment itself is disposed.

Error handling

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

```
} catch (GRBException e) {
  Console.WriteLine("Error code: " + e.ErrorCode + ". " + e.Message);
}
```

Building and running the example

The C# and Visual Basic example directories (`<installdir>/examples/c#` and `<installdir>/examples/vb`) contain a number of examples. We encourage you to browse and modify them in order to become more familiar with the Gurobi .NET interface. We also encourage you to read the Gurobi Example Tour for more information.

On Windows, you can use the `CS_examples_2015.sln` or `CS_examples_2017.sln` solution files in `<installdir>/examples/build` to build and run the example with Visual Studio 2015 or 2017, respectively. Clicking on the `mip1_cs` project, and then selecting *Run* from the *Build* menu will compile and run the example.

Alternatively, you can find a .NET Core 2 based project file `dotnetcore2.csproj` under `<installdir>/examples/build/DOTNETCore2` together with the `mip1_cs` example. By executing `dotnet run` in this directory you can build and run the `mip1_cs` example. (It is possible to exchange this example with any other example provided in `<installdir>/examples/c#` by replacing the file `mip1_cs.cs` with the corresponding `.cs` example file.)

Python Interface

The Gurobi Python interface can be used in a number of ways. It is the basis of our Interactive Shell, where it is typically used to work with existing models. It can also be used to write standalone programs that create and solve models, in much the same way that you would use our other language interfaces. The Gurobi distribution includes a Python interpreter and a basic set of Python modules. If you'd like additional capabilities, you can also install the Anaconda Python distribution, which includes both a graphical development environment (Spyder) and a notebook-style interface (Jupyter).

When comparing our Python interface against our other language interfaces, you will find that our Python interface provides a lot more options for building a model. You can work with individual variables and constraints, like you do in other object-oriented interfaces. You can work with matrices, like you do in our matrix-oriented interfaces. Our Python interface also includes a few higher-level constructs that allow you to build models using a more mathematical syntax, similar to the way you might work with a traditional modeling language.

This section will work through three Python modeling examples. The first will present a Python program that is similar to the C, C++, Java, and C# programs presented in previous sections. The second shows how to build models using matrices. The third demonstrates some of the higher-level modeling capabilities of our Python interface.

This section assumes that you are already familiar with the Python programming language, and that you have read the preceding section on the Gurobi Interactive Shell. If you would like to learn more about the Python language, we recommend that you visit the Python online tutorial.

As we noted, the Gurobi distribution includes all the tools you will need to run Python programs. However, if you would prefer to use your own Python installation, we also provide tools for installing the `gurobipy` module into your Python environment. You should refer to the instructions for building and running the examples for further details.

One big advantage of working within Python is that the Python language is popular and well supported. One aspect of this support is the breadth of powerful Python Integrated Development Environments (IDEs) that are available, most of which can be downloaded for free from the internet. This document includes instructions for setting up Gurobi for use within the Anaconda distribution. Anaconda greatly simplifies the task of installing Python packages, and it includes both a graphical development environment (Spyder) and a notebook-style interface (Jupyter). If you plan to do significant Python development, we recommend that you install Anaconda now. You will also find pointers to other useful Python tools there.

Important note for AIX users: due to limited Python support on AIX, our AIX port does not include the Interactive Shell or the Python interface. You can use the C, C++, or Java interfaces.

The Python example directory contains a number of examples. We encourage you to browse and modify them in order to become more familiar with the Gurobi Python interface. We also encourage

you to read the Gurobi Example Tour for more information.

12.1 Simple Python Example

This section will work through a simple Python example in order to illustrate the use of the Gurobi Python interface. The example builds a model, optimizes it, and outputs the optimal objective value.

Our example optimizes the following model:

$$\begin{array}{lrcrcrcl} \textbf{maximize} & x & + & y & + & 2\,z & & \\ \textbf{subject to} & x & + & 2\,y & + & 3\,z & \leq & 4 \\ & x & + & y & & & \geq & 1 \\ & & & \multicolumn{5}{c}{x, y, z \text{ binary}} \end{array}$$

Note that this is the same model that was modeled and optimized in the C Interface section.

Example mip1.py

This is the complete source code for our example (also available in `<installdir>/examples/python/mip1.py`)...

```
#!/usr/bin/env python3.7

# Copyright 2020, Gurobi Optimization, LLC

# This example formulates and solves the following simple MIP model:
#  maximize
#        x +   y + 2 z
#  subject to
#        x + 2 y + 3 z <= 4
#        x +   y       >= 1
#        x, y, z binary

import gurobipy as gp
from gurobipy import GRB

try:

    # Create a new model
    m = gp.Model("mip1")

    # Create variables
    x = m.addVar(vtype=GRB.BINARY, name="x")
    y = m.addVar(vtype=GRB.BINARY, name="y")
    z = m.addVar(vtype=GRB.BINARY, name="z")

    # Set objective
    m.setObjective(x + y + 2 * z, GRB.MAXIMIZE)

    # Add constraint: x + 2 y + 3 z <= 4
```

```
    m.addConstr(x + 2 * y + 3 * z <= 4, "c0")

    # Add constraint: x + y >= 1
    m.addConstr(x + y >= 1, "c1")

    # Optimize model
    m.optimize()

    for v in m.getVars():
        print('%s %g' % (v.varName, v.x))

    print('Obj: %g' % m.objVal)

except gp.GurobiError as e:
    print('Error code ' + str(e.errno) + ': ' + str(e))

except AttributeError:
    print('Encountered an attribute error')
```

Example details

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

The example begins by importing Gurobi functions and classes:

```
import gurobipy as gp
```

The first line makes all Gurobi functions and classes available through a `gp.` prefix (e.g., `gp.Model()`). The second makes everything in class `GRB` available without a prefix (e.g., `GRB.OPTIMAL`). You can also start a program with `from gurobipy import *` to remove the need for the `gp.` prefix, but if your program uses multiple Python modules it is usually preferred to access each through its own prefix.

In order for these commands to succeed, the Python application needs to know how to find the Gurobi functions and classes. Recall that you have three options here. The first is to use the Python files that are included with our distribution. You would run this example by typing `gurobi.bat mip1.py`. The second is to install the Anaconda Python distribution. The third is to install the Gurobi functions and classes into your own Python installation.

Creating the model

The first step in our example is to create a model. A Gurobi model holds a single optimization problem. It consists of a set of variables, a set of constraints, and the associated attributes (variable bounds, objective coefficients, variable integrality types, constraint senses, constraint right-hand side values, etc.). We start this example with an empty model object:

```
    # Create a new model
    m = gp.Model("mip1")
```

This function takes the desired model name as its argument.

Adding variables to the model

The next step in our example is to add variables to the model.

```
# Create variables
x = m.addVar(vtype=GRB.BINARY, name="x")
```

Variables are added through the `addVar()` method on a model object (or `addVars()` if you wish to add more than one at a time). A variable is always associated with a particular model.

Python allows you to pass arguments by position or by name. We've passed them by name here. Each variable gets a type (binary), and a name. We use the default values for the other arguments. Please refer to the online help (`help(Model.addVar)` in the Gurobi Shell) for further details on `addVar()`.

Setting the objective

The next step in the example is to set the optimization objective:

```
# Set objective
m.setObjective(x + y + 2 * z, GRB.MAXIMIZE)
```

The objective is built here using overloaded operators. The Python API overloads the arithmetic operators to allow you to build linear and quadratic expressions involving Gurobi variables.

The second argument indicates that the sense is maximization.

Note that while this simple example builds the objective in a single statement using an explicit list of terms, more complex programs will typically build it incrementally. For example:

```
obj = LinExpr();
obj += x;
obj += y;
obj += 2*z;
model.setObjective(obj, GRB.MAXIMIZE);
```

Adding constraints to the model

The next step in the example is to add the linear constraints. The first constraint is added here:

```
# Add constraint: x + 2 y + 3 z <= 4
m.addConstr(x + 2 * y + 3 * z <= 4, "c0")
```

As with variables, constraints are always associated with a specific model. They are created using the *addConstr()* method on the model object.

We again use overloaded arithmetic operators to build linear expressions. The comparison operators are also overloaded to make it easier to build constraints.

The second argument to `addConstr` gives the (optional) constraint name.

Again, this simple example builds the linear expression for the constraint in a single statement using an explicit list of terms. More complex programs will typically build the expression incrementally.

The second constraint is created in a similar manner:

```
# Add constraint: x + y >= 1
m.addConstr(x + y >= 1, "c1")
```

Optimizing the model

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

```
# Optimize model
m.optimize()
```

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

Reporting results - attributes

Once the optimization is complete, we can query the values of the attributes. In particular, we can query the `varName` and `x` variable attributes to obtain the name and solution value for each variable:

```
for v in m.getVars():
    print('%s %g' % (v.varName, v.x))
```

We can also query the `objVal` attribute on the model to obtain the objective value for the current solution:

```
print('Obj: %g' % m.objVal)
```

The names and types of all model, variable, and constraint attributes can be found in the online Python documentation. Type `help(GRB.Attr)` in the Gurobi Shell for details.

Error handling

Errors in the Gurobi Python interface are handled through the Python exception mechanism. In the example, all Gurobi statements are enclosed inside a `try` block, and any associated errors would be caught by the `except` block:

```
except gp.GurobiError as e:
    print('Error code ' + str(e.errno) + ': ' + str(e))

except AttributeError:
    print('Encountered an attribute error')
```

Running the example

When you run the example `gurobi.bat mip1.py`, you should see the following output:

```
Using license file c:\gurobi\gurobi.lic
Set parameter LogFile to value gurobi.log


Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 2 rows, 3 columns and 5 nonzeros
Model fingerprint: 0xb2adf8c4
Variable types: 0 continuous, 3 integer (3 binary)
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [1e+00, 2e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 4e+00]
Found heuristic solution: objective 2.0000000
Presolve removed 2 rows and 3 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.00 seconds
Thread count was 1 (of 4 available processors)

Solution count 2: 3

Optimal solution found (tolerance 1.00e-04)
Best objective 3.000000000000e+00, best bound 3.000000000000e+00, gap 0.0000%
x 1
y 0
z 1
Obj: 3
```

12.2 Python Matrix Example

You can also build optimization models in Python using dense and sparse matrices, much like you would in our MATLAB and R interfaces. We'll show an example that works through the same simple model from the first part of this section:

$$
\begin{array}{lrcrcrcl}
\textbf{maximize} & \mathrm{x} & + & \mathrm{y} & + & 2\,\mathrm{z} & & \\
\textbf{subject to} & \mathrm{x} & + & 2\,\mathrm{y} & + & 3\,\mathrm{z} & \leq & 4 \\
 & \mathrm{x} & + & \mathrm{y} & & & \geq & 1 \\
 & & & \multicolumn{3}{c}{\mathrm{x,\ y,\ z\ binary}} & &
\end{array}
$$

Example matrix1.py

This is the complete source code for our example (also available in `<installdir>/examples/python/matrix1.py`)...

```
#!/usr/bin/env python3.7

# Copyright 2020, Gurobi Optimization, LLC

# This example formulates and solves the following simple MIP model
# using the matrix API:
```

```
#  maximize
#        x +   y + 2 z
#  subject to
#        x + 2 y + 3 z <= 4
#        x +   y       >= 1
#        x, y, z binary

import numpy as np
import scipy.sparse as sp
import gurobipy as gp
from gurobipy import GRB

try:

    # Create a new model
    m = gp.Model("matrix1")

    # Create variables
    x = m.addMVar(shape=3, vtype=GRB.BINARY, name="x")

    # Set objective
    obj = np.array([1.0, 1.0, 2.0])
    m.setObjective(obj @ x, GRB.MAXIMIZE)

    # Build (sparse) constraint matrix
    data = np.array([1.0, 2.0, 3.0, -1.0, -1.0])
    row = np.array([0, 0, 0, 1, 1])
    col = np.array([0, 1, 2, 0, 1])

    A = sp.csr_matrix((data, (row, col)), shape=(2, 3))

    # Build rhs vector
    rhs = np.array([4.0, -1.0])

    # Add constraints
    m.addConstr(A @ x <= rhs, name="c")

    # Optimize model
    m.optimize()

    print(x.X)
    print('Obj: %g' % m.objVal)

except gp.GurobiError as e:
    print('Error code ' + str(e.errno) + ": " + str(e))

except AttributeError:
    print('Encountered an attribute error')
```

You will need to have both the NumPy package and the SciPy sparse matrix package in your Python environment to run this example. The easiest way to obtain a suitable Python environment is to install the Anaconda Python distribution.

Example details

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

The example begins by importing the Gurobi functions and classes, as well as the NumPy and SciPy packages:

```
import numpy as np
```

Gurobi Python applications should always start with the first line. If you want to use the matrix interface, the next two lines are needed as well.

Creating the model

The first step in our example is to create a model. We start with an empty model object:

```
# Create a new model
m = gp.Model("matrix1")
```

This function takes the desired model name as its argument.

Adding variables to the model

The next step adds a matrix variable to the model:

```
# Create variables
x = m.addMVar(shape=3, vtype=GRB.BINARY, name="x")
```

A matrix variable is added through the `addMVar()` method on a model object. In this case the matrix variable consists of a 1-D array of 3 binary variables. Variables are always associated with a particular model.

Setting the objective

The next step is to set the optimization objective:

```
obj = np.array([1.0, 1.0, 2.0])
m.setObjective(obj @ x, GRB.MAXIMIZE)
```

The objective is built here by computing a dot product between a constant vector and our matrix variable using the overloaded `@` operator. Note that the constant vector must have the same length as our matrix variable.

The second argument indicates that the sense is maximization.

Adding constraints to the model

The next step in the example is to add our two linear constraints. This is done by building a sparse matrix that captures the constraint matrix:

```
# Build (sparse) constraint matrix
data = np.array([1.0, 2.0, 3.0, -1.0, -1.0])
row = np.array([0, 0, 0, 1, 1])
col = np.array([0, 1, 2, 0, 1])
```

The matrix has two rows, one for each constraint, and three columns, one for each variable in our matrix variable. Note that we multiply the greater-than constraint by -1 to transform it to a less-than constraint.

We also capture the right-hand side in a NumPy array:

```
# Build rhs vector
```

We then use the overloaded `@` operator to build a linear matrix expression, and then use the overloaded less-than-or-equal operator to add two constraints (one for each row in the matrix expression):

```
# Add constraints
m.addConstr(A @ x <= rhs, name="c")
```

Optimizing the model

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

```
# Optimize model
m.optimize()
```

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

Reporting results - attributes

Once the optimization is complete, we can query the values of the attributes. In particular, we can query the `varName` and `x` variable attributes to obtain the name and solution value for each variable:

```
print(x.X)
```

We can also query the `objVal` attribute on the model to obtain the objective value for the current solution:

```
print('Obj: %g' % m.objVal)
```

The names and types of all model, variable, and constraint attributes can be found in the online Python documentation. Type `help(GRB.Attr)` in the Gurobi Shell for details.

Error handling

Errors in the Gurobi Python interface are handled through the Python exception mechanism. In the example, all Gurobi statements are enclosed inside a `try` block, and any associated errors would be caught by the `except` block:

```
except gp.GurobiError as e:
    print('Error code ' + str(e.errno) + ": " + str(e))

except AttributeError:
    print('Encountered an attribute error')
```

Running the example

When you run the example `gurobi.bat matrix1.py`, you should see the following output:

```
Using license file c:\gurobi\gurobi.lic
Set parameter LogFile to value gurobi.log


Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 2 rows, 3 columns and 5 nonzeros
Model fingerprint: 0xbc579ebb
Variable types: 0 continuous, 3 integer (3 binary)
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [1e+00, 2e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 4e+00]
Found heuristic solution: objective 2.0000000
Presolve removed 2 rows and 3 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.00 seconds
Thread count was 1 (of 4 available processors)

Solution count 2: 3

Optimal solution found (tolerance 1.00e-04)
Best objective 3.000000000000e+00, best bound 3.000000000000e+00, gap 0.0000%
[1. 0. 1.]
Obj: 3
```

12.3 Python Dictionary Example

In order to provide a gentle introduction to our interfaces, the examples so far have demonstrated only very basic capabilities. We will now attempt to demonstrate some of the power of our Python interface by describing a more complex example. This example is intended to capture most of the common ingredients of large, complex optimization models. Implementing this same example in another API would most likely have required hundreds of lines of code (ours is around 70 lines of Python code).

We'll need to present a few preliminaries before getting to the example itself. You'll need to learn a bit about the Python language, and we'll need to describe a few custom classes and functions. Our intent is that you will come away from this section with an appreciation for the power and flexibility of this interface. It can be used to create quite complex models using what we believe

are very concise and natural modeling constructs. Our goal with this interface has been to provide something that feels more like a mathematical modeling language than a programming language API.

If you'd like to dig a bit deeper into the Python language constructs described here, we recommend that you visit the online Python tutorial.

Motivation

At the heart of any optimization model lies a set of decision variables. Finding a convenient way to store and access these variables can often represent the main challenge in implementing the model. While the variables in some models map naturally to simple programming language constructs (e.g., `x[i]` for contiguous integer values `i`), other models can present a much greater challenge. For example, consider a model that optimizes the flow of multiple different commodities through a supply network. You might have a variable `x['Pens', 'Denver', 'New York']` that captures the flow of a manufactured item (pens in this example) from Denver to New York. At the same time, you might *not* want to have a variable `x['Pencils', 'Denver', 'Seattle']`, since not all combinations of commodities, source cities, and destination cities represent valid paths through the network. Representing a sparse set of decision variables in a typical programming language can be cumbersome.

To compound the challenge, you typically need to build constraints that involve subsets of these decision variables. For example, in our network flow model you might want to put an upper bound on the total flow that enters a particular city. You could certainly collect the relevant decision variables by iterating over all possible cities and selecting only those variables that capture possible flow from that source city into the desired destination city. However, this is clearly wasteful if not all origin-destination pairs are valid. In a large network problem, the inefficiency of this approach could lead to major performance issues. Handling this efficiently can require complex data structures.

The Gurobi Python interface has been designed to make the issues we've just described quite easy to manage. We'll present a specific example of how this is done shortly. Before we do, though, we'll need to describe a few important Python constructs: `lists`, `tuples`, `dictionaries`, `list comprehension`, and `generator expressions`. These are standard Python concepts that are particularly important in our interface. We'll also introduce the `tuplelist` and `tupledict` classes, which are custom classes that we've added to the Gurobi Python interface.

A quick reminder: you can consult the online Python documentation for additional information on any of the Python data structures mentioned here.

Lists and Tuples

The `list` data structure is central to most Python programs; Gurobi Python programs are no exception. We'll also rely heavily on a similar data structure, the `tuple`. Tuples are crucial to providing efficient and convenient access to Gurobi decision variables in Gurobi Python programs. The difference between a list and a tuple is subtle but important. We'll discuss it shortly.

Lists and tuples are both just ordered collections of Python objects. A list is created and dis-

played as a comma-separated list of member objects, enclosed in square brackets. A tuple is similar, except that the member objects are enclosed in parenthesis. For example, `[1, 2, 3]` is a list, while `(1, 2, 3)` is a tuple. Similarly, `['Pens', 'Denver', 'New York']` is a list, while `('Pens', 'Denver', 'New York')` is a tuple.

You can retrieve individual entries from a list or tuple using square brackets and zero-based indices:

```
gurobi> l = [1, 2.0, 'abc']
gurobi> t = (1, 2.0, 'abc')
gurobi> print(l[0])
1
gurobi> print(t[1])
2.0
gurobi> print(l[2])
abc
```

What's the difference between a list and a tuple? A tuple is *immutable*, meaning that you can't modify it once it has been created. By contrast, you can add new members to a list, remove members, change existing members, etc. This immutable property allows you to use tuples as indices for `dictionaries`.

Dictionaries

A Python `dictionary` allows you to map arbitrary `key` values to pieces of data. Any *immutable* Python object can be used as a key: an integer, a floating-point number, a string, or even a tuple.

To give an example, the following statements create a dictionary `x`, and then associates a value `1` with key `('Pens', 'Denver', 'New York')`

```
gurobi> x = {}  # creates an empty dictionary
gurobi> x[('Pens', 'Denver', 'New York')] = 1
gurobi> print(x[('Pens', 'Denver', 'New York')])
1
```

Python allows you to omit the parenthesis when accessing a dictionary using a tuple, so the following is also valid:

```
gurobi> x = {}
gurobi> x['Pens', 'Denver', 'New York'] = 2
gurobi> print(x['Pens', 'Denver', 'New York'])
2
```

We've stored integers in the dictionary here, but dictionaries can hold arbitrary objects. In particular, they can hold Gurobi decision variables:

```
gurobi> x['Pens', 'Denver', 'New York'] = model.addVar()
gurobi> print(x['Pens', 'Denver', 'New York'])
<gurobi.Var *Awaiting Model Update*>
```

To initialize a dictionary, you can of course simply perform assignments for each relevant key:

```
gurobi> values = {}
gurobi> values['zero'] = 0
gurobi> values['one'] = 1
gurobi> values['two'] = 2
```

You can also use the Python dictionary initialization construct:

```
gurobi> values = { 'zero': 0, 'one': 1, 'two': 2 }
gurobi> print(values['zero'])
0
gurobi> print(values['one'])
1
```

We have included a utility routine in the Gurobi Python interface that simplifies dictionary initialization for a case that arises frequently in mathematical modeling. The `multidict` function allows you to initialize one or more dictionaries in a single statement. The function takes a dictionary as its argument, where the value associated with each key is a list of length `n`. The function splits these lists into individual entries, creating `n` separate dictionaries. The function returns a list. The first result is the list of shared key values, followed by the `n` individual dictionaries:

```
gurobi> names, lower, upper = multidict({ 'x': [0, 1], 'y': [1, 2], 'z': [0, 3] })
gurobi> print(names)
['x', 'y', 'z']
gurobi> print(lower)
{'x': 0, 'y': 1, 'z': 0}
gurobi> print(upper)
{'x': 1, 'y': 2, 'z': 3}
```

Note that you can also apply this function to a dictionary where each key maps to a scalar value. In that case, the function simply returns the list of keys as the first result, and the original dictionary as the second.

You will see this function in several of our Python examples.

List comprehension and generator expressions

List comprehension and generator expressions are important Python features that allows you to do implicit enumeration in a concise fashion. To give a simple example, the following list comprehension builds a list containing the squares of the numbers from 1 through 5:

```
gurobi> [x*x for x in [1, 2, 3, 4, 5]]
[1, 4, 9, 16, 25]
```

A generator expression is very similar, but it is used to generate an `Iterable` (something that can be iterated over). For example, suppose we want to compute the sum of the squares of the numbers from 1 through 5. We could use list comprehension to build the list, and then pass that list to `sum`. However, it is simpler and more efficient to use a generator expression:

```
gurobi> sum(x*x for x in [1, 2, 3, 4, 5])
55
```

A generator expression can be used whenever a method accepts an `Iterable` argument (something that can be iterated over). For example, most Python methods that accept a `list` argument (the most common type of `Iterable`) will also accept a generator expression.

Note that there's a Python routine for creating a contiguous list of integers: `range`. The above would typically be written as follows:

```
gurobi> sum(x*x for x in range(1,6))
```

Details on the `range` function can be found here.

List comprehension and generator expressions can both contain more than one `for` clause, and one or more `if` clauses. The following example builds a list of tuples containing all `x,y` pairs where `x` and `y` are both less than 4 and `x` is less than `y`:

```
gurobi> [(x,y) for x in range(4) for y in range(4) if x < y]
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
```

Note that the `for` statements are executed left-to-right, and values from one can be used in the next, so a more efficient way to write the above is:

```
gurobi> [(x,y) for x in range(4) for y in range(x+1, 4)]
```

Generator expressions are used extensively in our Python examples, primarily in the context of the `addConstrs` method.

The tuplelist class

The next important item we would like to discuss is the `tuplelist` class. This is a custom sub-class of the Python `list` class that is designed to allow you to efficiently build sub-lists from a list of tuples. To be more specific, you can use the `select` method on a `tuplelist` object to retrieve all tuples that match one or more specified values in specific fields.

Let us give a simple example. We'll begin by creating a simple `tuplelist` (by passing a list of tuples to the constructor):

```
gurobi> l = tuplelist([(1, 2), (1, 3), (2, 3), (2, 4)])
```

To select a sub-list where particular tuple entries match desired values, you specify the desired values as arguments to the `select` method. The number of arguments to `select` is equal to the number of entries in the members of the `tuplelist` (they should all have the same number of entries). You can provide a list argument to indicate that multiple values are acceptable in that position in the tuple, or a `'*'` string to indicate that any value is acceptable.

Each tuple in our example contains two entries, so we can perform the following selections:

```
gurobi> print(l.select(1, '*'))
<gurobi.tuplelist (2 tuples, 2 values each):
 ( 1 , 2 )
 ( 1 , 3 )
>
```

```
gurobi> print(l.select('*', 3))
<gurobi.tuplelist (2 tuples, 2 values each):
 ( 1 , 3 )
 ( 2 , 3 )
>
gurobi> print(l.select('*', [2, 4]))
<gurobi.tuplelist (2 tuples, 2 values each):
 ( 1 , 2 )
 ( 2 , 4 )
>
gurobi> print(l.select(1, 3))
<gurobi.tuplelist (1 tuples, 2 values each):
 ( 1 , 3 )
>
gurobi> print(l.select('*', '*'))
<gurobi.tuplelist (4 tuples, 2 values each):
 ( 1 , 2 )
 ( 1 , 3 )
 ( 2 , 3 )
 ( 2 , 4 )
>
```

You may have noticed that similar results could have been achieved using list comprehension. For example:

```
gurobi> print(l.select(1, '*'))
<gurobi.tuplelist (2 tuples, 2 values each):
 ( 1 , 2 )
 ( 1 , 3 )
>
gurobi> print([(x,y) for x,y in l if x == 1])
[(1, 2), (1, 3)]
```

The problem is that the latter statement considers every member in the list, which can be quite inefficient for large lists. The `select` method builds internal data structures that make these selections quite efficient.

Note that `tuplelist` is a sub-class of `list`, so you can use the standard `list` methods to access or modify a `tuplelist`:

```
gurobi> print(l[1])
(1,3)
gurobi> l += [(3, 4)]
gurobi> print(l)
<gurobi.tuplelist (5 tuples, 2 values each):
 ( 1 , 2 )
 ( 1 , 3 )
 ( 2 , 3 )
 ( 2 , 4 )
 ( 3 , 4 )
>
```

The tupledict class

The final important preliminary we would like to discuss is the `tupledict` class. This is a custom sub-class of the Python `dict` class that allows you to efficiently work with subsets of Gurobi variable objects. To be more specific, you can use the `sum` and `prod` methods on a `tupledict` object to easily and concisely build linear expressions. The keys for a `tupledict` are stored as a `tuplelist`, so the same `select` syntax can be used to choose subsets of entries. Specifically, by associating a tuple with each Gurobi variable, you can efficiently create expressions that contain a subset of matching variables. For example, using the `sum` method on a `tupledict` object, you could easily build an expression that captures the sum over all Gurobi variables for which the first field of the corresponding tuple is equal to 3 (using `x.sum(3, '*')`).

While you can directly build your own `tupledict`, the Gurobi interface provides an `addVars` method that adds one Gurobi decision variable to the model for each tuple in the input argument(s) and returns the result as a `tupledict`. Let us give a simple example. We'll begin by constructing a list of tuples, and then we'll create a set of Gurobi variables that are indexed using this list:

```
gurobi> l = list([(1, 2), (1, 3), (2, 3), (2, 4)])
gurobi> d = model.addVars(l, name="d")
gurobi> model.update()
```

The `addVars` method will create variables `d(1,2)`, `d(1,3)`, `d(2,3)`, and `d(2,4)`. Note that the `name` argument is used to name the resulting variables, but it only gives the prefix for the name - the names are subscripted by the tuple keys (so the variables would be named `d[1,2]`, `d[1,3]`, etc.). The final call to `update` synchronizes certain internal data structures; this detail can be safely ignored for now.

You can then use this `tupledict` to build linear expressions. For example, you could do:

```
gurobi> sum(d.select(1, '*'))
```

The `select` method returns a list of Gurobi variables where the first field of the associated tuple is 1. The Python `sum` statement then creates a linear expression that captures the sum of these variables. In this case, that expression would be `d(1,2) + d(1,3)`. Similarly, `sum(d.select('*', 3))` would give `d(1,3) + d(2,3)`. As with a `tuplelist`, you use a `'*'` string to indicate that any value is acceptable in that position in the tuple.

The `tupledict` class includes a method that simplifies the above. Rather than `sum(d.select('*', 3))`, you can use `d.sum('*', 3)` instead.

The `tupledict` class also includes a `prod` method, for cases where your linear expression has coefficients that aren't all `1.0`. Coefficients are provided through a `dict` argument. They are indexed using the same tuples as the `tupledict`. For example, given a `dict` named `coeff` with two entries: `coeff(1,2)` = 5 and `coeff(2,3)` = 7, a call to `d.prod(coeff)` would give the expression `5 d(1,2) + 7 d(2,3)`. You can also include a filter, so `d.prod(coeff, 2, '*')` would give just `7 d(2,3)`.

Note that `tupledict` is a sub-class of `dict`, so you can use the standard `dict` methods to access or modify a `tupledict`:

```
gurobi> print(d[1,3])
```

```
<gurobi.Var d[1,3]>
gurobi> d[3, 4] = 0.3
gurobi> print(d[3, 4])
0.3
gurobi> print(d.values())
dict_values([<gurobi.Var d[1,2]>, 0.3, <gurobi.Var d[1,3]>, <gurobi.Var d[2,3]>, <gurobi.Var d[2,4]>])
```

In our upcoming network flow example, once we've built a `tupledict` that contains a variable for each valid commodity-source-destination combination on the network (we'll call it `flows`), we can create a linear expression that captures the total flow on all arcs that empty into a specific destination city as follows:

```
gurobi> inbound = flows.sum('*', '*', 'New York')
```

We now present an example that illustrates the use of all of the concepts discussed so far.

netflow.py example

Our example solves a multi-commodity flow model on a small network. In the example, two commodities (Pencils and Pens) are produced in two cities (Detroit and Denver), and must be shipped to warehouses in three cities (Boston, New York, and Seattle) to satisfy given demand. Each arc in the transportation network has a cost associated with it, and a total capacity.

This is the complete source code for our example (also available in `<installdir>/examples/python/netflow.py`)...

```
#!/usr/bin/env python3.7

# Copyright 2020, Gurobi Optimization, LLC

# Solve a multi-commodity flow problem.  Two products ('Pencils' and 'Pens')
# are produced in 2 cities ('Detroit' and 'Denver') and must be sent to
# warehouses in 3 cities ('Boston', 'New York', and 'Seattle') to
# satisfy demand ('inflow[h,i]').
#
# Flows on the transportation network must respect arc capacity constraints
# ('capacity[i,j]'). The objective is to minimize the sum of the arc
# transportation costs ('cost[i,j]').

import gurobipy as gp
from gurobipy import GRB

# Base data
commodities = ['Pencils', 'Pens']
nodes = ['Detroit', 'Denver', 'Boston', 'New York', 'Seattle']

arcs, capacity = gp.multidict({
    ('Detroit', 'Boston'):   100,
    ('Detroit', 'New York'):  80,
    ('Detroit', 'Seattle'):  120,
    ('Denver',  'Boston'):   120,
    ('Denver',  'New York'): 120,
    ('Denver',  'Seattle'):  120})
```

```
# Cost for triplets commodity-source-destination
cost = {
    ('Pencils', 'Detroit', 'Boston'):   10,
    ('Pencils', 'Detroit', 'New York'): 20,
    ('Pencils', 'Detroit', 'Seattle'):  60,
    ('Pencils', 'Denver',  'Boston'):   40,
    ('Pencils', 'Denver',  'New York'): 40,
    ('Pencils', 'Denver',  'Seattle'):  30,
    ('Pens',    'Detroit', 'Boston'):   20,
    ('Pens',    'Detroit', 'New York'): 20,
    ('Pens',    'Detroit', 'Seattle'):  80,
    ('Pens',    'Denver',  'Boston'):   60,
    ('Pens',    'Denver',  'New York'): 70,
    ('Pens',    'Denver',  'Seattle'):  30}

# Demand for pairs of commodity-city
inflow = {
    ('Pencils', 'Detroit'):   50,
    ('Pencils', 'Denver'):    60,
    ('Pencils', 'Boston'):   -50,
    ('Pencils', 'New York'): -50,
    ('Pencils', 'Seattle'):  -10,
    ('Pens',    'Detroit'):   60,
    ('Pens',    'Denver'):    40,
    ('Pens',    'Boston'):   -40,
    ('Pens',    'New York'): -30,
    ('Pens',    'Seattle'):  -30}

# Create optimization model
m = gp.Model('netflow')

# Create variables
flow = m.addVars(commodities, arcs, obj=cost, name="flow")

# Arc-capacity constraints
m.addConstrs(
    (flow.sum('*', i, j) <= capacity[i, j] for i, j in arcs), "cap")

# Equivalent version using Python looping
# for i, j in arcs:
#   m.addConstr(sum(flow[h, i, j] for h in commodities) <= capacity[i, j],
#               "cap[%s, %s]" % (i, j))


# Flow-conservation constraints
m.addConstrs(
    (flow.sum(h, '*', j) + inflow[h, j] == flow.sum(h, j, '*')
        for h in commodities for j in nodes), "node")

# Alternate version:
# m.addConstrs(
#   (gp.quicksum(flow[h, i, j] for i, j in arcs.select('*', j)) + inflow[h, j] ==
#     gp.quicksum(flow[h, j, k] for j, k in arcs.select(j, '*'))
#     for h in commodities for j in nodes), "node")
```

```
# Compute optimal solution
m.optimize()

# Print solution
if m.status == GRB.OPTIMAL:
    solution = m.getAttr('x', flow)
    for h in commodities:
        print('\nOptimal flows for %s:' % h)
        for i, j in arcs:
            if solution[h, i, j] > 0:
                print('%s -> %s: %g' % (i, j, solution[h, i, j]))
```

netflow.py example details

Let us now walk through the example, line by line, to understand how it achieves the desired result of computing the optimal network flow. As with the simple Python example presented earlier, this example begins by importing the Gurobi functions and classes:

```
import gurobipy as gp
```

We then create a few lists that contain model data:

```
# Base data
commodities = ['Pencils', 'Pens']
nodes = ['Detroit', 'Denver', 'Boston', 'New York', 'Seattle']

arcs, capacity = gp.multidict({
    ('Detroit', 'Boston'):   100,
    ('Detroit', 'New York'):  80,
    ('Detroit', 'Seattle'):  120,
    ('Denver',  'Boston'):   120,
    ('Denver',  'New York'): 120,
    ('Denver',  'Seattle'):  120})
```

The model works with two commodities (Pencils and Pens), and the network contains 5 nodes and 6 arcs. We initialize `commodities` and `nodes` as simple Python lists. We use the Gurobi `multidict` function to initialize `arcs` (the `list` of keys) and `capacity` (a `dictionary`).

The model also requires cost data for each commodity-arc pair:

```
# Cost for triplets commodity-source-destination
cost = {
    ('Pencils', 'Detroit', 'Boston'):   10,
    ('Pencils', 'Detroit', 'New York'): 20,
    ('Pencils', 'Detroit', 'Seattle'):  60,
    ('Pencils', 'Denver',  'Boston'):   40,
    ('Pencils', 'Denver',  'New York'): 40,
    ('Pencils', 'Denver',  'Seattle'):  30,
    ('Pens',    'Detroit', 'Boston'):   20,
    ('Pens',    'Detroit', 'New York'): 20,
    ('Pens',    'Detroit', 'Seattle'):  80,
    ('Pens',    'Denver',  'Boston'):   60,
    ('Pens',    'Denver',  'New York'): 70,
    ('Pens',    'Denver',  'Seattle'):  30}
```

Once this dictionary has been created, the cost of moving commodity `h` from node `i` to `j` can be queried as `cost[(h,i,j)]`. Recall that Python allows you to omit the parenthesis when using a tuple to index a dictionary, so this can be shortened to just `cost[h,i,j]`.

A similar construct is used to initialize node demand data:

```
# Demand for pairs of commodity-city
inflow = {
    ('Pencils', 'Detroit'):   50,
    ('Pencils', 'Denver'):    60,
    ('Pencils', 'Boston'):   -50,
    ('Pencils', 'New York'): -50,
    ('Pencils', 'Seattle'):  -10,
    ('Pens',    'Detroit'):   60,
    ('Pens',    'Denver'):    40,
    ('Pens',    'Boston'):   -40,
    ('Pens',    'New York'): -30,
    ('Pens',    'Seattle'):  -30}
```

Building a multi-dimensional array of variables

The next step in our example (after creating an empty `Model` object) is to add variables to the model. The variables are created using `addVars`, and are returned in a `tupledict` which we'll call `flow`:

```
# Create optimization model
m = gp.Model('netflow')

# Create variables
flow = m.addVars(commodities, arcs, obj=cost, name="flow")
```

The first, positional arguments to `addVars` give the index set. In this case, we'll be indexing `flow` by `commodities` and `arcs`. In other words, `flow[c,i,j]` will capture the flow of commodity `c` from node `i` to node `j`. Note that `flow` only contains variables for source, destination pairs that are present in `arcs`.

Arc capacity constraints

We begin with a straightforward set of constraints. The sum of the flow variables on an arc must be less than or equal to the capacity of that arc:

```
# Arc-capacity constraints
m.addConstrs(
    (flow.sum('*', i, j) <= capacity[i, j] for i, j in arcs), "cap")
```

Note that this one statement uses several of the concepts that were introduced earlier in this section.

The first concept used here is the `sum` method on `flow`, which is used to create a linear expression over a subset of the variables in the `tupledict`. In particular, it is summing over all commodities (the `'*'` in the first field) associated with an edge between a pair of cities `i` and `j`.

The second concept used here is a generator expression, which iterates over all arcs in the network. Specifically, this portion of the statement...

```
for i,j in arcs
```

indicates that we are iterating over every edge in `arcs`. In each iteration, `i` and `j` will be populated using the corresponding values from a tuple in `arcs`. In a particular iteration, `flow.sum('*',i,j)` will be computed using those specific values, as will `capacity[i,j]`.

The third thing to note is that we're passing the result as an argument to `addConstrs`. This method will create a set of Gurobi constraints, one for each iteration of the generator expression.

The final thing to note is that the last argument gives the base for the constraint name. The `addConstrs` method will automatically append the corresponding indices for each constraint. Thus, for example, the name of the constraint that limits flow from Denver to Boston will be `cap[Denver,Boston]`.

Note that if you prefer to do your own looping, you could obtain the equivalent behavior with the following loop:

```
for i,j in arcs:
    m.addConstr(sum(flow[h,i,j] for h in commodities) <= capacity[i,j],
                "cap[%s,%s]" % (i, j))
```

Flow conservation constraints

The next set of constraints are the flow conservation constraints. They require that, for each commodity and node, the sum of the flow into the node plus the quantity of external inflow at that node must be equal to the sum of the flow out of the node:

```
# Flow-conservation constraints
m.addConstrs(
    (flow.sum(h, '*', j) + inflow[h, j] == flow.sum(h, j, '*')
        for h in commodities for j in nodes), "node")
```

This call to `addConstrs` is similar to the previous one, although a bit more complex. We invoke the `sum` method on a `tupledict`, wrapped inside of a generator expression, to add one linear constraint for each commodity-node pair. In this instance, we call `sum` twice, and we use a generator expression that contains of a pair of `for` loops, but the basic concepts remain the same.

Results

Once we've added the model constraints, we call `optimize` and then output the optimal solution:

```
# Compute optimal solution
m.optimize()

# Print solution
if m.status == GRB.OPTIMAL:
    solution = m.getAttr('x', flow)
    for h in commodities:
        print('\nOptimal flows for %s:' % h)
        for i, j in arcs:
            if solution[h, i, j] > 0:
                print('%s -> %s: %g' % (i, j, solution[h, i, j]))
```

If you run the example `gurobi.bat netflow.py`, you should see the following output:

```
Using license file c:\gurobi\gurobi.lic
Set parameter LogFile to value gurobi.log


Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 16 rows, 12 columns and 36 nonzeros
Model fingerprint: 0xf10778ba
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+01, 8e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+01, 1e+02]
Presolve removed 16 rows and 12 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    5.5000000e+03   0.000000e+00   2.000000e+01      0s
Extra one simplex iteration after uncrush
       1    5.5000000e+03   0.000000e+00   0.000000e+00      0s

Solved in 1 iterations and 0.00 seconds
Optimal objective  5.500000000e+03

Optimal flows for Pencils:
Detroit -> Boston: 50
Denver -> New York: 50
Denver -> Seattle: 10

Optimal flows for Pens:
Detroit -> Boston: 30
Detroit -> New York: 30
Denver -> Boston: 10
Denver -> Seattle: 30
```

12.4 Building and running the examples

Python is an interpreted language, so no explicit compilation step is required to run the examples. For Windows platforms, you can simply type the following in the Gurobi Python example directory (`<installdir>/examples/python`):

```
gurobi.bat mip1.py
```

If you are a Python user and wish to use Gurobi from within your own Python environment, we recommend that you use the Anaconda Python distribution. Anaconda includes a number of very useful packages, and the Gurobi Anaconda package simplifies the installation process.

You can also install the `gurobipy` module directly into other Python environments. The steps for doing this depend on your platform. On Windows, you can double-click on the `pysetup` program in the Gurobi `<installdir>/bin` directory. This program will prompt you for the location of your Python installation; it handles all of the details of the installation.

Once `gurobipy` is successfully installed, you can type `python mip1.py` (more generally, you can type `from gurobipy import *` in your Python environment).

This section describes the Gurobi MATLAB interface. We begin with information on how to set up Gurobi for use within MATLAB. An example of how to use the MATLAB interface follows.

Setting up Gurobi for MATLAB

To begin, you'll need to tell MATLAB where to find the Gurobi routines. We've provided a script to assist you with this. The Gurobi MATLAB setup script, `gurobi_setup.m`, can be found in the `<installdir>/matlab` directory of your Gurobi installation (the default `<installdir>` for Gurobi 9.0.3 is `c:\gurobi903\win64` for 64-bit Windows). To get started, type the following commands within MATLAB to change to the `matlab` directory and call `gurobi_setup`:

```
>> cd c:\gurobi903\win64\matlab
>> gurobi_setup
```

You will need to be careful that the MATLAB binary and the Gurobi package you install both use the same instruction set. You need to install the 64-bit version of MATLAB to use Gurobi. This is particularly important on Windows systems, where the error messages that result from instruction set mismatches can be quite cryptic.

Example

Let us now turn our attention to an example of using Gurobi to solve a simple MIP model. Our example optimizes the following model:

$$
\begin{array}{lrcrcrcl}
\textbf{maximize} & x & + & y & + & 2\,z & & \\
\textbf{subject to} & x & + & 2\,y & + & 3\,z & \leq & 4 \\
 & x & + & y & & & \geq & 1 \\
 & & & \multicolumn{5}{c}{x, y, z \text{ binary}}
\end{array}
$$

Note that this is the same model that was modeled and optimized in the C Interface section.

This is the complete source code for our example (also available in `<installdir>/examples/matlab/mip1.m`)...

```
function mip1()
% Copyright 2020, Gurobi Optimization, LLC
% This example formulates and solves the following simple MIP model:
%  maximize
%        x +   y + 2 z
%  subject to
%        x + 2 y + 3 z <= 4
%        x +   y       >= 1
%        x, y, z binary

names = {'x'; 'y'; 'z'};

model.A = sparse([1 2 3; 1 1 0]);
```

```
model.obj = [1 1 2];
model.rhs = [4; 1];
model.sense = '<>';
model.vtype = 'B';
model.modelsense = 'max';
model.varnames = names;

gurobi_write(model, 'mip1.lp');

params.outputflag = 0;

result = gurobi(model, params);

disp(result);

for v=1:length(names)
    fprintf('%s %d\n', names{v}, result.x(v));
end

fprintf('Obj: %e\n', result.objval);
end
```

Example details

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

Building the model

The example begins by building an optimization model. The data associated with an optimization model must be stored in a MATLAB `struct`. Fields in this struct contain the different parts of the model. The most fundamental fields are: The constraint matrix (`A`), the objective vector (`obj`), the right-hand side vector (`rhs`), and the constraint sense vector (`sense`). Among these, only the constraint matrix is mandatory, and default values are substituted for all other model fields in case they are missing.

The example uses the built-in `sparse` function to build the constraint matrix `A`. The Gurobi MATLAB interface only accepts sparse matrices as input. If you have a dense matrix, use `sparse` to convert it to a sparse matrix before passing it to our interface.

In addition to the fields discussed above, this example sets two more fields: `modelsense` and `vtype`. The former is used to indicate the sense of the objective function. The default is minimization, so we've set the field equal to `'max'` to indicate that we would like to maximize the specified objective. The `vtype` field is used to indicate the types of the variables in the model. In our example, all variables are binary (`'B'`). Note that our interface allows you to specify a scalar value for the `sense` and `vtype` arguments. The Gurobi interface will expand that scalar to a constant array of the appropriate length. In this example, the scalar value `'B'` will be expanded to an array of length 3, containing one `'B'` value for each column of `A`.

Modifying Gurobi parameters

The next statements create a `struct` variable that will be used to modify two Gurobi parameters:

```
params.outputflag = 0;
params.resultfile = 'mip1.lp';
```

In this example, we set the Gurobi `OutputFlag` parameter to 0 in order to shut off Gurobi output. We also set the *ResultFile* parameter to request that Gurobi produce a file as output (in this case, an `LP` format file that contains the optimization model). The Gurobi MATLAB interface allows you to set as many Gurobi parameters as you like. The field names in the parameter structure simply need to match Gurobi parameter names, and the values of the fields should be set to the desired parameter values. Please consult the *Parameters* section of the Gurobi Reference Manual for a complete list of all Gurobi parameters.

Solving the model

The next statement is where the actual optimization occurs:

```
result = gurobi(model, params);
```

We pass the `model` and the optional list of parameter changes to the `gurobi()` function. It computes an optimal solution to the specified model and returns the computed result.

Printing the solution

The `gurobi()` function returns a `struct` as its result. This struct contains a number of fields, where each field contains information about the computed solution. The available fields depend on the result of the optimization, the type of model that was solved (LP, QP, QCP, SOCP, or MIP), and the algorithm used to solve the model. The returned `struct` will always contain a `status` field, which indicates whether Gurobi was able to compute an optimal solution to the model. You should consult the *Status Codes* section of the Gurobi Reference Manual for a complete list of all possible status codes. If Gurobi was able to find a solution to the model, the return value will also include `objval` and `x` fields. The former gives the objective value for the computed solution, and the latter is the computed solution vector (one entry per column of the constraint matrix). For continuous models, we will also return dual information (reduced costs and dual multipliers), and possibly an optimal basis.

In our example, we simply print the optimal objective value (`result.objval`) and the optimal solution vector (`result.x`).

Running the example

The Gurobi MATLAB examples can be found in the `<installdir>/examples/matlab/` directory of your Gurobi installation (the default `<installdir>` for Gurobi 9.0.3 is `c:\gurobi903\win64 for 64-bit Windows`). To run one of the examples, first change to this directory in MATLAB, then type its name into the MATLAB prompt. For example, to run example `mip1`, you would say:

```
>> cd c:\gurobi903\win64\examples\matlab
>> mip1
```

If Gurobi was successfully set up for use in MATLAB, you should see the following output in the command window:

```
         status: 'OPTIMAL'
    versioninfo: [1x1 struct]
        runtime: 3.2401e-04
         objval: 3
```

```
               x: [3x1 double]
           slack: [2x1 double]
    poolobjbound: 3
            pool: [1x2 struct]
          mipgap: 0
        objbound: 3
       objboundc: 3
       itercount: 0
    baritercount: 0
       nodecount: 0

x 1
y 0
z 1
Obj: 3.000000e+00
```

From all this data we only use the fields `objval` and `x` in our example. Please refer to the reference manual for a complete description of all the other output fields.

In order to get more familiar with the Gurobi MATLAB interface, we encourage you to browse through the files in the MATLAB example directory. Often these examples can be used as starting points for your own optimization projects.

This section describes the Gurobi R interface. We begin with information on how to set up Gurobi for use within R. An example of how to use the R interface follows.

Installing the R Package

To begin, you'll need to install the Gurobi package in R. The R command for doing this is:

```
install.packages('<R-package-file>', repos=NULL)
```

The R package file can be found in the `<installdir>/R` directory of your Gurobi installation. For a default installation of Gurobi 9.0.3, the command would be:

```
install.packages('c:/gurobi903/win64/R/gurobi_9.0-3.zip', repos=NULL)
```

You will need to adjust the path to match your installation directory and version.

You will need to be careful that the R binary and the Gurobi package you install both use the same instruction set. To use the Gurobi R package, you will need to use the 64-bit version of R. This is particularly important on Windows systems, where the error messages that result from instruction set mismatches can be quite cryptic.

If you are using R from RStudio Server, and you get an error indicating that R is unable to load the Gurobi DLL or shared object, you may need to set the `rsession-ld-library-path` entry in the server config file. Please consult the RStudio documentation for more information.

Example

Let us now turn our attention to an example of using Gurobi to solve a simple MIP model. Our example optimizes the following model:

$$
\begin{array}{lrcrcrcl}
\textbf{maximize} & x & + & y & + & 2\,z & & \\
\textbf{subject to} & x & + & 2\,y & + & 3\,z & \leq & 4 \\
 & x & + & y & & & \geq & 1 \\
 & & & \multicolumn{5}{c}{x, y, z \text{ binary}}
\end{array}
$$

Note that this is the same model that was modeled and optimized in the C Interface section.

This is the complete source code for our example (also available in `<installdir>/examples/R/mip.R`)...

```
# Copyright 2020, Gurobi Optimization, LLC
#
# This example formulates and solves the following simple MIP model:
#  maximize
#        x +   y + 2 z
#  subject to
#        x + 2 y + 3 z <= 4
```

```
#         x +    y        >= 1
#         x, y, z binary

library(gurobi)

model <- list()

model$A           <- matrix(c(1,2,3,1,1,0), nrow=2, ncol=3, byrow=T)
model$obj         <- c(1,1,2)
model$modelsense  <- 'max'
model$rhs         <- c(4,1)
model$sense       <- c('<', '>')
model$vtype       <- 'B'

params <- list(OutputFlag=0)

result <- gurobi(model, params)

print('Solution:')
print(result$objval)
print(result$x)

# Clear space
rm(model, result, params)
```

Example details

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

The example begins by importing the Gurobi package (`library('gurobi')`). R programs that call Gurobi must include this line.

Building the model

The example now builds an optimization model. The data associated with an optimization model must be stored in a single list variable. Named components in this list contain the different parts of the model. The most fundamental named components are: The constraint matrix (`A`), the objective vector (`obj`), the right-hand side vector (`rhs`), and the constraint sense vector (`sense`). Among these, only the constraint matrix is mandatory, and default values are substituted for all other model fields in case they are missing. A model variable can also include optional components (e.g., the objective sense `modelsense`).

In our example, we use the built-in R `matrix` function to build the constraint matrix `A`. `A` is stored as a dense matrix here. You can also store `A` as a sparse matrix, using either the `sparse_triplet_matrix` function from the `slam` package or the `sparseMatrix` class from the `Matrix` package. Sparse input matrices are illustrated in the `lp2.R` example.

Subsequent statements populate other components of the `model` variable, including the objective vector, the right-hand side vector, and the constraint sense vector. In each case, we use the built-in `c` function to initialize the array arguments.

In addition to the mandatory components, this example also sets two optional components: `modelsense` and `vtype`. The former is used to indicate the sense of the objective function. The default is minimization, so we've set the component equal to `'max'` to indicate that we would like to maximize the specified

objective. The `vtype` component is used to indicate the types of the variables in the model. In our example, all variables are binary (`'B'`). Note that our interface allows you to specify a scalar value for any array argument. The Gurobi interface will expand that scalar to a constant array of the appropriate length. In this example, the scalar value `'B'` will be expanded to an array of length 3, containing one `'B'` value for each column of `A`.

One important note about default variable bounds: the convention in math programming is that a variable will by default have a lower bound of 0 and an infinite upper bound. If you'd like your variables to have different bounds, you'll need to provide them explicitly.

Modifying Gurobi parameters

The next statement creates a list variable that will be used to modify a Gurobi parameter:

```
params <- list(OutputFlag=0)
```

In this example, we wish to set the Gurobi `OutputFlag` parameter to 0 in order to shut off Gurobi output. The Gurobi R interface allows you to pass a list of the Gurobi parameters you would like to change. Please consult the *Parameters* section of the Gurobi Reference Manual for a complete list of all Gurobi parameters.

Solving the model

The next statement is where the actual optimization occurs:

```
result <- gurobi(model, params)
```

We pass the `model` and the optional list of parameter changes to the `gurobi()` function. It computes an optimal solution to the specified model and returns the computed result.

Printing the solution

The `gurobi()` function returns a list as its result. This list contains a number of components, where each component contains information about the computed solution. The available components depend on the result of the optimization, the type of model that was solved (LP, QP, SOCP, or MIP), and the algorithm used to solve the model. This result list will always contain an integer `status` component, which indicates whether Gurobi was able to compute an optimal solution to the model. You should consult the *Status Codes* section of the Gurobi Reference Manual for a complete list of all possible status codes. If Gurobi was able to find a solution to the model, the return value will also include `objval` and `x` components. The former gives the objective value for the computed solution, and the latter is the computed solution vector (one entry per column of the constraint matrix). For continuous models, we will also return dual information (reduced costs and dual multipliers), and possibly an optimal basis.

In our example, we simply print the optimal objective value (`result$objval`) and the optimal solution vector (`result$x`).

Running the example

To run one of the R examples provided with the Gurobi distribution, you can use the `source` command in R. For example, if you are running R from the Gurobi R examples directory, you can say:

```
> source('mip.R')
```

If the Gurobi package was successfully installed, you should see the following output:

```
[1] "Solution:"
[1] 3
[1] 1 0 1
```

The R example directory `<installdir>/examples/R` contains a number of examples. We encourage you to browse and modify them in order to become more familiar with the Gurobi R interface.

Recommended Reading

The very basic introduction to mathematical programming and mathematical modeling in this document barely scratches the surface of this very broad and rich field. We've collected a set of recommended books here that provide more information on various aspects of math programming.

If you want more information on the algorithms and mathematics underlying the solution of linear programming problems, we recommend Introduction to Linear Optimization by Bertsimas, Tsitsiklis, and Tsitsiklis, or Linear Programming: Foundations and Extensions by R. Vanderbei. For a detailed treatment of interior-point methods for linear programming, we recommend Primal-Dual Interior-Point Methods by S. Wright.

For more information on the algorithms and mathematics underlying the solution of mixed-integer programming problems, we recommend Integer Programming by L. Wolsey.

For an introduction to the process of creating mathematical programming representations of business problems, we recommend Model Building in Mathematical Programming by H.P. Williams.

Installing the Anaconda Python distribution

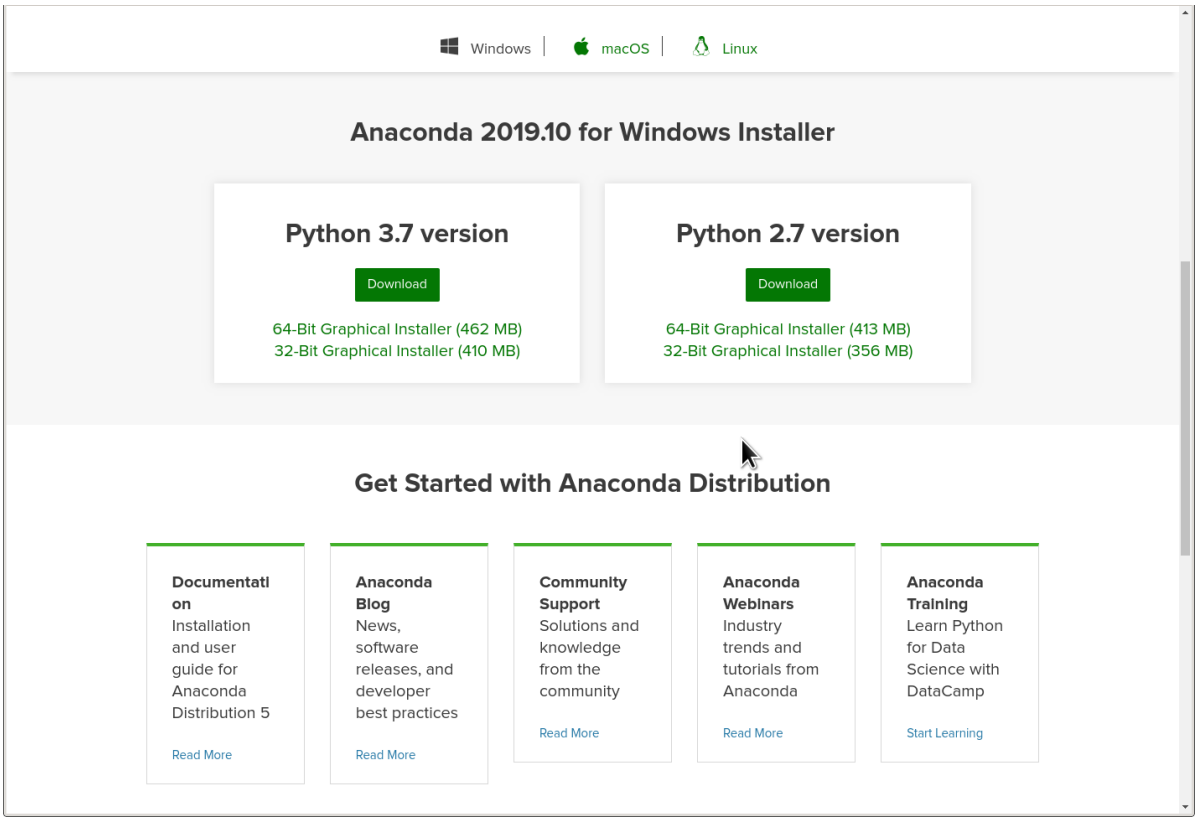
The Gurobi distribution includes a Python interpreter and a basic set of Python modules. While these are sufficient for building and running simple optimization models, they provide just a glimpse of the wealth of tools and modules that are available for Python. This section guides you through the steps involved in installing Anaconda, a widely-used Python platform that includes an Integrated Development Environment (Spyder), a notebook-style interface (Jupyter), and a broad set of Python modules. These tools can significantly increase the interactivity and productivity of your Python model building experience.

Before we begin, we should note that Anaconda isn't your only choice in Python IDEs. Popular alternatives include Eric, iep, PyCharm, and PyDev. We won't be covering the details of installing these other options for use with Gurobi, but the Anaconda instructions that follow should provide a good outline for the steps involved. We've found that Anaconda provides a nice balance between power and complexity, but we realize that people may look for different things in their Python environments.

You will also find instructions for installing Anaconda Python at the Get Anaconda page on our website.

Step 1: Download and Install Anaconda

The first step is to download and install Anaconda. You can find detailed instructions here. Be sure to check the box `Add Anaconda to my PATH environment variable` when prompted during the installation process. Otherwise, Step 2 may not work.



Gurobi supports the 64-bit versions of Python 2.7, 3.6, and 3.7. Click on the download button for the version you want. Once the download has been completed, run the Anaconda installer.

Step 2: Install Gurobi into Anaconda

The next step is to install the Gurobi package into Anaconda. You do this by first adding the Gurobi channel to your Anaconda channels and then installing the `gurobi` package from this channel.

From a terminal window issue the following command to add the Gurobi channel to your default search list

```
conda config --add channels http://conda.anaconda.org/gurobi
```

Now issue the following command to install the Gurobi package

```
conda install gurobi
```

You can remove the Gurobi package at any time by issuing the command

```
conda remove gurobi
```

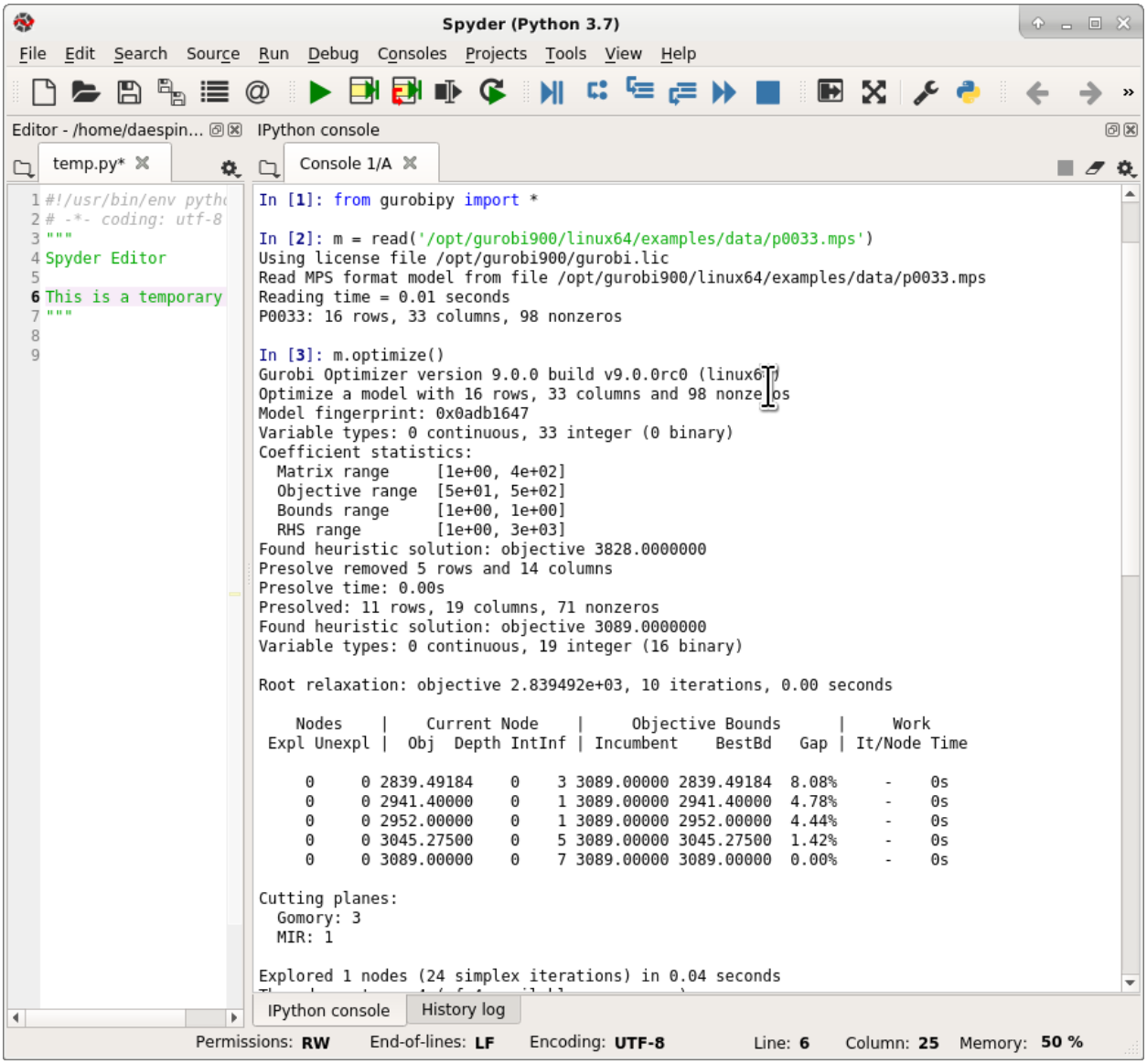
Step 3: Install a Gurobi License

The third step is to install a Gurobi license (if you haven't already done so).

You are now ready to use Gurobi from within Anaconda. Your next step is to launch either the Spyder IDE or Jupyter.

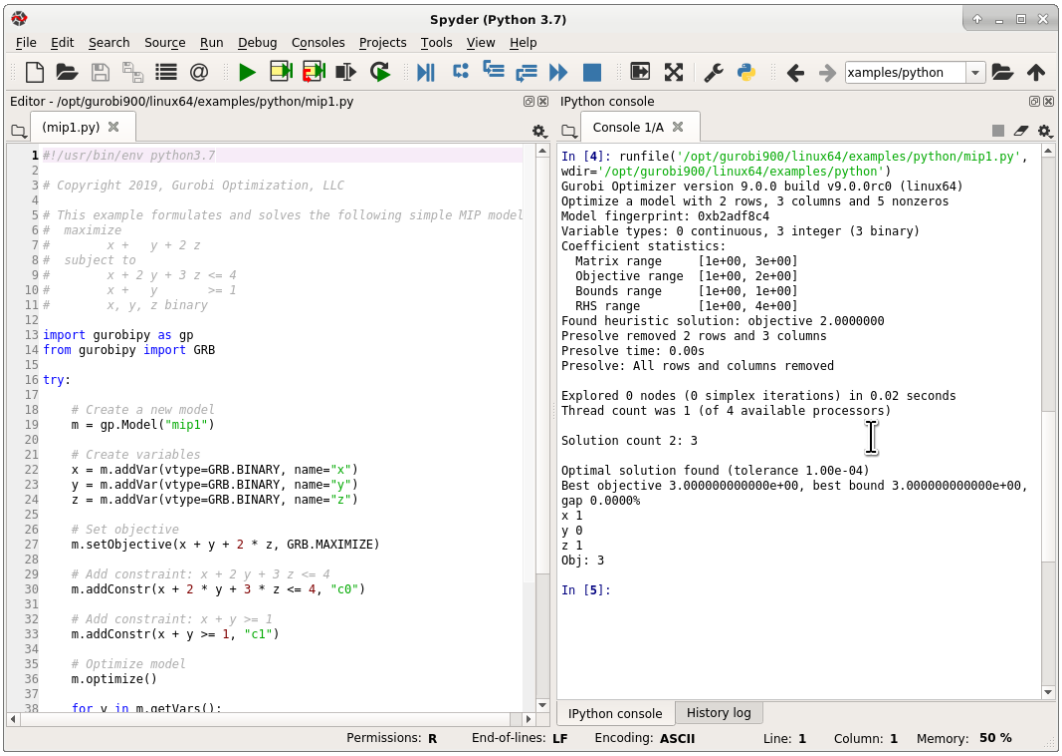
16.1 Using the Spyder IDE

You can launch the `Spyder` Python IDE, which is included in Anaconda Python, from the Windows Program menu (the installer should have placed it in the Anaconda folder).Gurobi Interactive Shell commands can be typed directly into the Spyder `Console` window:

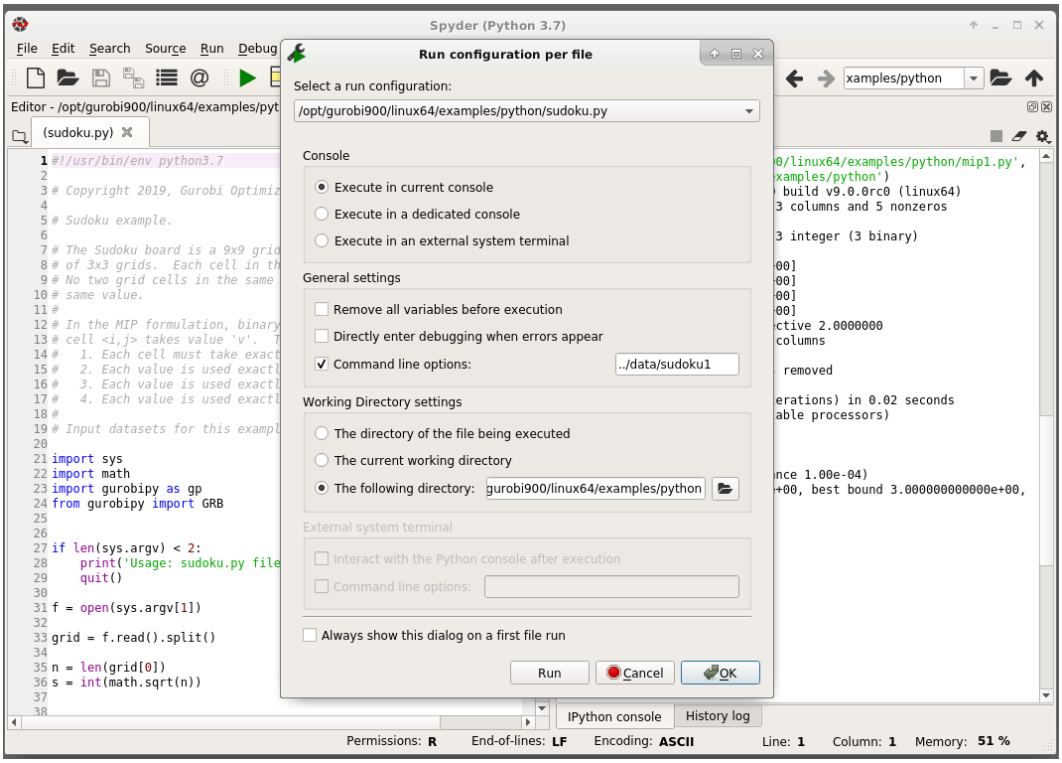


Note that a general-purpose Python IDE like Spyder requires one extra step that isn't required when you launch the Gurobi shell from the Gurobi icon or by using the `gurobi.bat` command: you must manually load the Gurobi module by typing `from gurobipy import *` (or `import gurobipy as gp`) before issuing any Gurobi commands.

You can also use Spyder to run any of the Gurobi examples. For example, if you use `Open` under the `File` menu to open Gurobi example `mip1.py`, and then click on the `Run` icon (the green triangle), you should see:



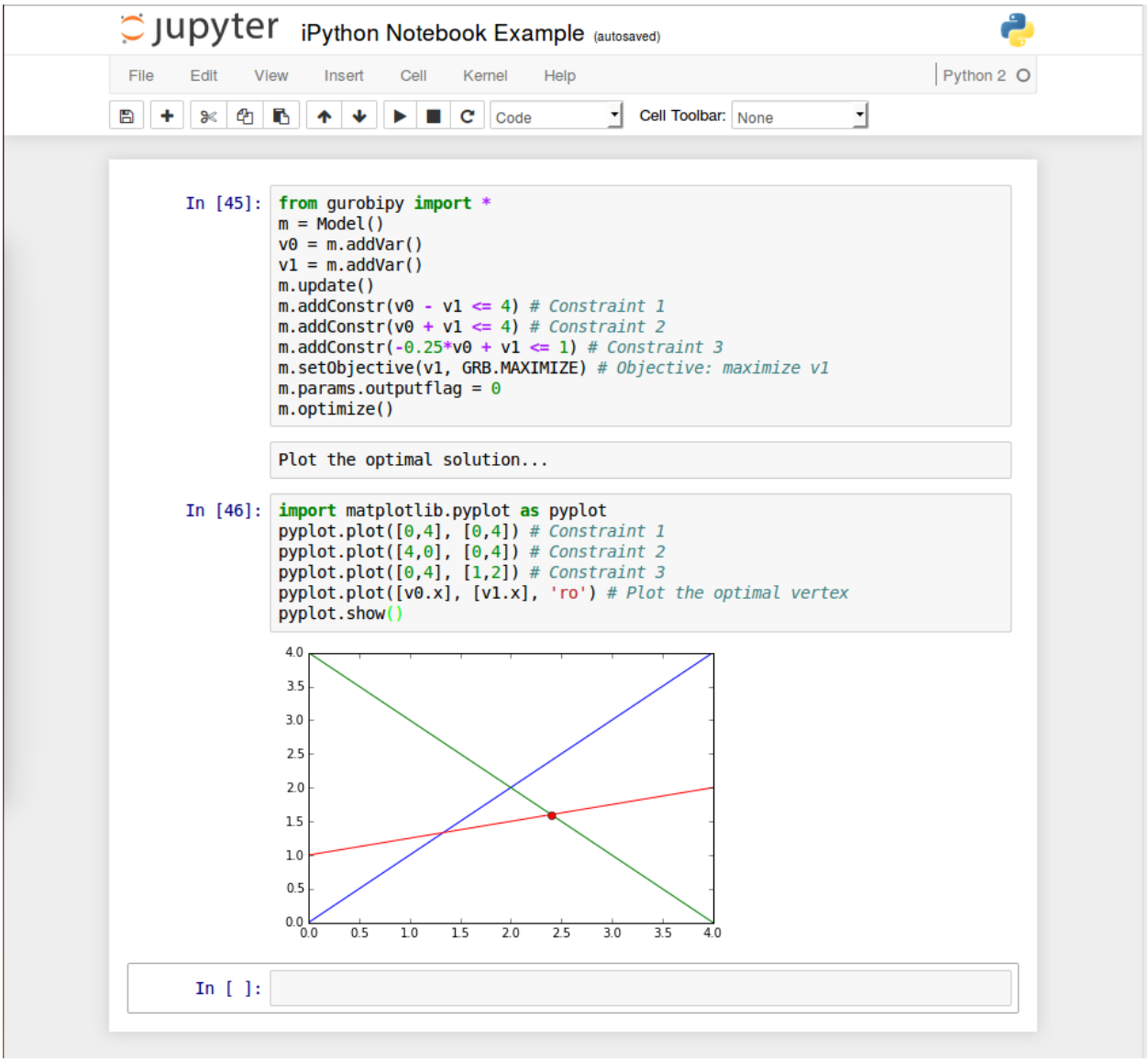
Some Gurobi examples require command-line arguments. Those can be input from the `Configure...` item of the `Run` menu. For example, to run the `sudoku.py` example with file `sudoku1` as input ...



Anaconda Python includes all of the Python modules used in the Gurobi examples, but feel free to explore the vast library of additional Python modules. A list of available modules can be found at the PyPI site. You can use the `conda` command to install additional modules. Type `conda` into a terminal window with no arguments to get additional information on this command.

16.2 Using Jupyter

You can launch `Jupyter`, which is included in Anaconda Python, from the Windows Program menu (the installer should have placed it in the Anaconda folder).You can create a new notebook by clicking on the `New` icon (in the upper right) and choosing one of the `Notebook` options. Once your new notebook starts, you can type standard Python commands or Gurobi Interactive Shell commands directly into the `In` window:



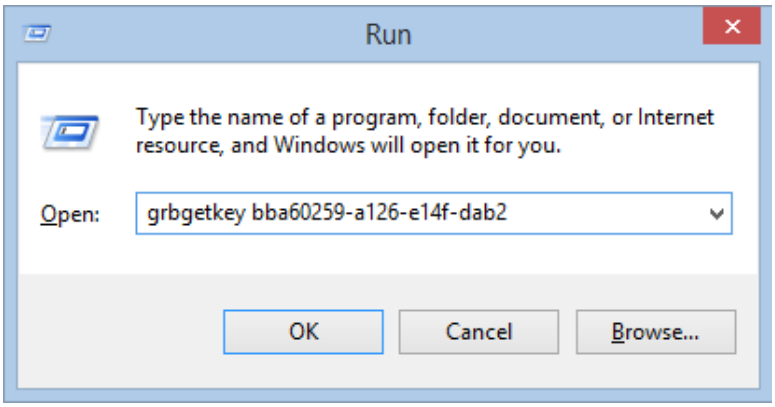
Our simple example shows a set of commands that create and solve a simple linear programming model, and then

plot the resulting constraints and the computed optimal vertex.

For those of you who aren't familiar with notebook-style interfaces, they allow you to mix executable code, text, and graphics to create a self-documenting stream of results. Notebooks can be saved and continued later, which make them particularly well suited for prototyping and experimentation.

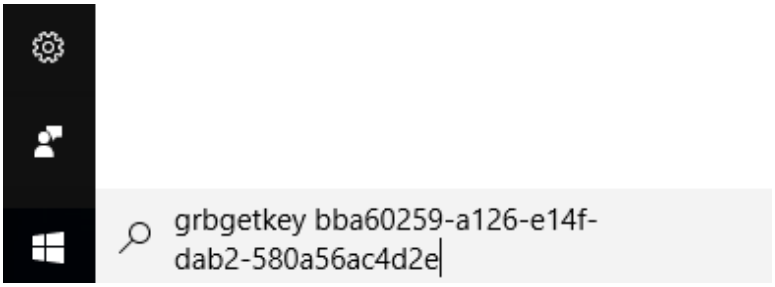
Setting up and using Gurobi on a Windows system sometimes requires you to type command-line commands. This can be unfamiliar to Windows users, so this section provides a bit of help.

Windows provides a few different options for inputting commands. For simple commands, you can open a *Run* window, by pressing the *Start* key and the letter *R* simultaneously:

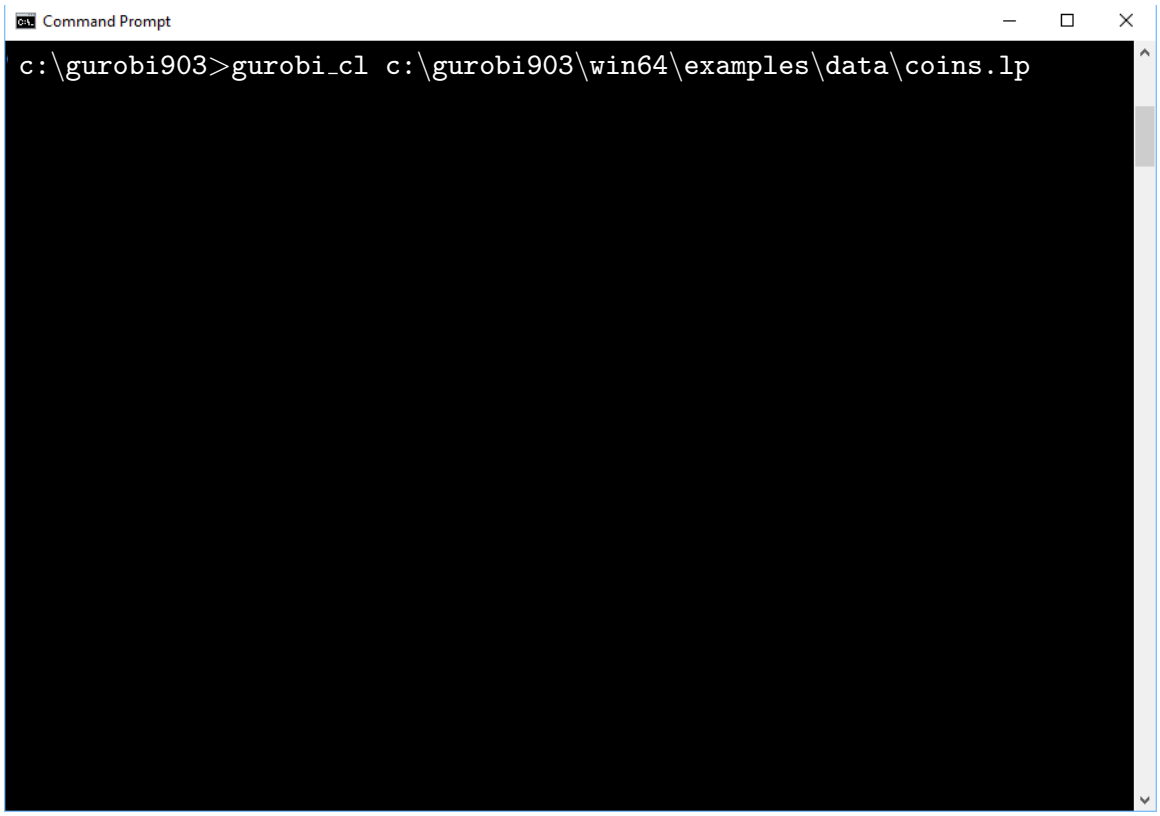


A command can be typed or pasted directly into this window.

You can also type or paste commands directly into the *Search* box (towards the bottom left of the desktop on Windows 10):



While these two interfaces are useful for issuing simple commands, they aren't suited for extensive interactive usage. If you want to launch Gurobi command-line commands and see the resulting output, for example, you'll need a *Console* window (also known as a *Command Prompt* window or just a *cmd* window). To launch a Console window, type `cmd` into either the *Run* window or the *Search* box (as described above).



This window will remain open, displaying the output from the previous command and waiting for the next command, until you close it.

File Overview

This section briefly describes the purposes of the more important files in the Gurobi distribution. Note that the list below may not precisely agree with your installation. We've omitted a few less important files. In addition, a few file names depend on the exact version of the Gurobi optimizer that you installed.

The following files and directories are created in your installation directory (typically `c:\gurobi903\win64` for the 64-bit Windows distribution):

- EULA.doc - Gurobi End User License Agreement - Microsoft Word format
- EULA.pdf - Gurobi End User License Agreement - PDF format
- ReleaseNotes.html - release notes
- bin
  - gurobi90.netstandard20.dll - .NET Standard 2.0 interface
  - gurobi90.netstandard20.xml - .NET Standard 2.0 interface documentation
  - Gurobi90.NET.XML - .NET Framework interface documentation
  - Gurobi90.NET.dll - .NET Framework interface
  - GurobiJni90.dll - Java JNI wrapper
  - grbcluster.exe - Compute Server command-line tool
  - grb_ts.exe - Gurobi Token Server executable
  - grbgetkey.exe - retrieves your Gurobi license key from the Gurobi key server
  - grbprobe.exe - probes system details (typically not used)
  - grbtune.exe - parameter tuning tool
  - gurobi.bat - starts the Gurobi interactive shell
  - gurobi.env - sample parameter initialization file
  - gurobi90.dll - Gurobi native DLL (used by all Gurobi interfaces)
  - gurobi90_light.dll - light Gurobi native DLL (no support for Compute Server or Instant Cloud)
  - gurobi_cl.exe - simple command-line binary
- docs
  - examples - Example Tour - HTML (open index.html in this directory)
  - examples.pdf - Example Tour - PDF
  - quickstart - Quick Start guide - HTML (open index.html in this directory)
  - quickstart_windows.pdf - Quick Start guide - PDF
  - refman - Reference Manual - HTML (open index.html in this directory)
  - refman.pdf - Reference Manual - PDF
  - remoteservices - Remote Services Reference Manual - HTML (open index.html in this directory)
  - remoteservices.pdf - Remote Services Reference Manual - PDF
- examples
  - build - Makefile for C, C++, C#, Java, and Python examples
  - c - source code for C examples

  - c# - source code for C# examples
  - c++ - source code for C++ examples
  - data - data files for examples
  - dotnetcore - Makefile and .NET Core App files for C# examples
  - java - source code for Java examples
  - matlab - source code for MATLAB examples
  - python - source code for Python examples
  - R - source code for R examples
  - vb - source code for Visual Basic examples (for Windows)
- include
  - gurobi_c.h - C include file
  - gurobi_c++.h - C++ include file
- lib
  - gurobi.jar - Java interface
  - gurobi-javadoc.jar - Javadoc documentation for our Java interface
  - gurobi.py - Python startup file
  - gurobi90.lib - Gurobi library import file
  - gurobi_c++md2015.lib - C++ interface (when using -MD compiler switch with Visual Studio 2015)
  - gurobi_c++md2017.lib - C++ interface (when using -MD compiler switch with Visual Studio 2017)
  - gurobi_c++mdd2015.lib - C++ interface (when using -MDd compiler switch with Visual Studio 2015)
  - gurobi_c++mdd2017.lib - C++ interface (when using -MDd compiler switch with Visual Studio 2017)
  - gurobi_c++mt2015.lib - C++ interface (when using -MT compiler switch with Visual Studio 2015)
  - gurobi_c++mt2017.lib - C++ interface (when using -MT compiler switch with Visual Studio 2017)
  - gurobi_c++mtd2015.lib - C++ interface (when using -MTd compiler switch with Visual Studio 2015)
  - gurobi_c++mtd2017.lib - C++ interface (when using -MTd compiler switch with Visual Studio 2017)
- matlab - Gurobi MATLAB interface
- R - R Gurobi package
- python27 - Python 2.7 files used by the interactive shell and the Python interface (no need to look inside this directory)
- python35 - Python 3.5 files (no need to look inside this directory)
- python36 - Python 3.6 files (no need to look inside this directory)
- src
  - build - Makefile for Gurobi C++ interface
  - cpp - Source for Gurobi C++ interface

The following files and directories are created in your Remote Services installation (typically `c:/gurobi_server903/win64` for the 64-bit Windows distribution):

- EULA.pdf - Gurobi End User License Agreement - PDF format
- ReleaseNotes.html - release notes
- bin
  - grbcluster.exe - Compute Server command-line tool
  - grb_rs.exe - Gurobi Remote Services executable

  - grb_rs.cnf - Default configuration file for the Remote Services
  - grb_rs_aws.cnf - Example configuration file for the Remote Services deployment with AWS
  - grb_rsr.exe - Gurobi Remote Services Router executable
  - grb_rsr.cnf - Default configuration file for the Remote Services Router
  - grb_ts.exe - Gurobi Token Server executable
  - grbgetkey.exe - retrieves your Gurobi license key from the Gurobi key server
  - grbprobe.exe - probes system details (typically not used)
  - data
    * files - Temporary data files for job execution
    * runtimes - Worker executables for supported versions
- docs
  - remoteservices - Remote Services Reference Manual - HTML (open index.html in this directory)
  - remoteservices.pdf - Remote Services Reference Manual - PDF
- resources - Internal resource files for web UI