GUROBI OPTIMIZER QUICK START GUIDE

Version 5.6, Copyright © 2013, Gurobi Optimization, Inc.

Contents

Introduction

Welcome to the Gurobi Optimizer Quick Start Guide! This document provides a basic introduction to the Gurobi™ Optimizer. It begins with a Software Installation Guide and information on How to Obtain and Install a Gurobi License. It continues with an example that shows you how to create a simple optimization model and solve it with the Gurobi Command Line. It then continues with a discussion of the Gurobi Interactive Shell. We suggest that all users read these first four sections.

Depending on the programming language from which you intend to use the Gurobi Optimizer, you may also wish to read the sections devoted to the Gurobi C Interface, C++ Interface, Java® Interface, Microsoft® .NET Interface, Python® Interface, MATLAB® Interface, or R Interface. Our Python interface will be of particular interest to those who wish to build models using higher-level modeling constructs.

This document continues with a File Overview, which gives a quick overview of the files that are included in the Gurobi distribution.

Additional resources

Once you are done with the Quick Start Guide, you can find additional information in a number of places:

- If you are familiar with mathematical modeling and want to dive right into the details of how to use Gurobi from one of our programming language APIs, you should consult the Gurobi Reference Manual.

- If you would like a tour of the various examples we provide with Gurobi, you should consult the Gurobi Example Tour.

- If you would like to learn more about mathematical programming or mathematical 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, you can post it to the Gurobi Google Group. If you have a current maintenance contract with us, you can send your question to support@gurobi.com.

For most users, your next step is to move on to our Software Installation Guide.

Software Installation Guide

Before using the Gurobi Optimizer, you'll need to install it on your computer. If you haven't already done so, please download the installation files from our download page. Simply select your platform (64-bit Windows, 64-bit Linux, etc.), and click on the *Download* button. Make a note of the name and location of the downloaded file.

The next installation step depends on your platform. Detailed instructions for Windows, Linux, and Mac OS follow.

Installation - Windows

To install the Gurobi optimizer on your Windows machine, double-click on the Gurobi installer that you downloaded from our website (e.g., `Gurobi-5.6.0-win32.msi` for the 32-bit version of Gurobi 5.6.0). You may have selected *Run* when downloading the installer, in which case you've already run the installer and don't need to do it again.

By default, the installer will place the Gurobi 5.6.0 files in directory `c:\gurobi560\win32` (or `c:\gurobi560\win64` for 64-bit Windows installs). The installer gives you the option to change the installation target. We'll refer to the installation directory as `<installdir>`.

Installation for all users

By default, Gurobi is installed for use by the current Windows user only. If you would like to install Gurobi for all users, you may do so 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-5.6.0-win32.msi allusers=1
```

You should normally only install the Gurobi version that is targeted to your platform (32-bit or 64-bit Windows). You do have the option of installing both on the same machine, but note that the Gurobi installer sets a few Windows environment variables that will point to one particular version. This can cause some confusion, particularly when a program asks Windows to load the Gurobi DLL. If you understand the implications, then feel free to install both.

Helpful tools

If you would like to work with compressed files from within the Gurobi Optimizer, we recommend that you also install `gzip` and/or `7zip`. They can be downloaded from www.gzip.org and www.7-zip.org, respectively.

Next steps

Once installation is complete, you should see a Gurobi desktop shortcut that can be used to launch the Gurobi Interactive Shell. You shouldn't try to launch Gurobi quite yet. Doing so will produce a lengthy error message indicating that you haven't yet installed a license key.

You are now ready to proceed to the section on How to Obtain and Install a Gurobi License.

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

Installation - Linux

The first step in installing the Gurobi Optimizer on a Linux system is to choose a destination directory. We recommend `/opt` for a shared installation, but other directories will work as well.

Once a destination directory has been chosen, the next step is to copy the Gurobi distribution to the destination directory and extract the contents. Extraction is done with the following command:

```
tar xvfz gurobi5.6.0_linux64.tar.gz
```

This command will create a sub-directory `gurobi560/linux64` that contains the complete Gurobi distribution, and your `<installdir>` would be `/opt/gurobi560/linux64`.

The Gurobi Optimizer makes use of several executable files. In order to allow these files to be found when needed, you will have to modify a few environment variables:

- `GUROBI_HOME` should point to your `<installdir>`.

- `PATH` should be extended to include `<installdir>/bin`.

- `LD_LIBRARY_PATH` should be extended to include `<installdir>/lib`.

Users of the `bash` shell would add the following lines to their `.bashrc` files...

```
export GUROBI_HOME="/opt/gurobi560/linux64"
export PATH="${PATH}:${GUROBI_HOME}/bin"
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${GUROBI_HOME}/lib"
```

Users of the `csh` shell would add the following lines to their `.cshrc` files...

```
setenv GUROBI_HOME /opt/gurobi560/linux64
setenv PATH ${PATH}:${GUROBI_HOME}/bin
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${GUROBI_HOME}/lib
```

If `LD_LIBRARY_PATH` is not already set, you would use the following instead:

```
export LD_LIBRARY_PATH="${GUROBI_HOME}/lib"
```

or

```
setenv LD_LIBRARY_PATH ${GUROBI_HOME}/lib
```

These paths should be adjusted to reflect your chosen `<installdir>`.

Next steps

You are now ready to proceed to the section on How to Obtain and Install a Gurobi License.

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

Installation - Mac OS

To install the Gurobi Optimizer on your Mac, double-click on the appropriate Gurobi installer (e.g., `gurobi5.6.0_mac64.pkg` for Gurobi 5.6.0) and follow the prompts. By default, the installer will place the Gurobi 5.6.0 files in `/Library/gurobi560/mac64`.

You are now ready to proceed to the section on How to Obtain and Install a Gurobi License.

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

How to Obtain and Install a Gurobi License

The Gurobi Optimizer requires a license to run. We support several different license types. Your first step in setting up your license is to figure out what type you need (or perhaps already have).

If you are looking to use a free trial or academic license, you can create one yourself on our website.

If you have purchased a license from us, that license should be visible through the Current tab of the *Licenses* page of our website. If you see it there, your next step is to retrieve the license key. Otherwise, contact us at `license@gurobi.com`.

If you are planning to use Gurobi as a client of a machine that is already set up as a Gurobi token server or compute server, you can create a client license yourself.

If you need to set up a Gurobi token server or Gurobi compute server, your first step is to retrieve your license key. Once you have done that, then you should consult the instructions for setting up a token server or for setting up a compute server, as appropriate.

If you already have a license key (stored in a `gurobi.lic` file), you can proceed to testing the license.

Creating a new trial or academic license

If you wish to use a free trial or academic license, you can create the license yourself. To do so, visit either the Free Trial or the Free Academic tab on the Licenses page at our website. First accept the license agreement, and then click on *Request License*. Your new license will be visible immediately, under the Current tab. You can create as many trial or academic licenses as you like.

Your next step is to get a license key.

Retrieving a license key

Once a license is visible under the Current tab, clicking on the *License ID* will produce a page like the following:

The next step is to associate this license with your machine by obtaining a *License Key*.

To obtain a Gurobi license key for your machine, you'll need to run the `grbgetkey` command on the machine on which you would like to run the Gurobi Optimizer. Note that the machine must be connected to the Internet in order to run this command. If you are using a free academic license, your machine must be connected from a recognized academic domain. Note that an Internet connection is not required once you have obtained your license key.

The exact command to run for a specific license is indicated at the bottom of the license page (e.g., `grbgetkey 253e22f3-...`). We recommend that you use copy-paste to grab the entire `grbgetkey` command from our website. On a Windows system, you can paste the command directly into the Windows *Run* box. Press the *Start* key and the letter *R* simultaneously to open this box. On Windows Vista and Windows 7, you can also paste the command into the *Search* box (available under the *Start* button). This is demonstrated in the Windows 7 example shown here: Paste the command into this box and then hit *Enter* to run the command. On a Linux machine, you
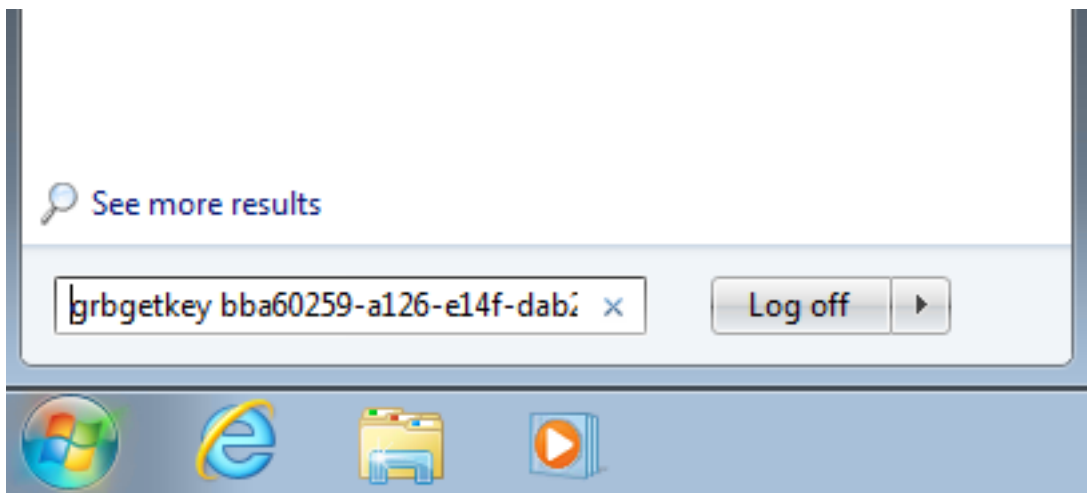
would paste this command into a shell window. On a Mac system, you would paste the command into a *Terminal* window.

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 occured, `grbgetkey` will ask you for the name of the directory in which to store your license key file. You should see a prompt that looks like the following

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


In which directory would you like to store the Gurobi license key file?
[hit Enter to store it in c:/Users/Jones]:
```

You can store the license file anywhere, but we recommend that you accept the default location (by hitting *Enter*). You can learn more about using a non-default license file location.

If you are installing a paid license and your computer isn't connected to the Internet, we also offer a manual installation process. You'll find manual installation instructions at the bottom of the license page (by following the link labeled *click here for additional instructions*).

If you are using a free academic license, `grbgetkey` will try to validate your academic license.

Next steps

If you are using one of our floating license types (including a single-use license), you'll also need to set up the token server.

If you are setting up a Gurobi compute server, you'll need set up the compute server.

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

Where to store your license key file

When you run 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, on a Windows system, Gurobi Optimizer 5.6.0 will also look in `c:\gurobi` and `c:\gurobi560`. On a Linux system, it will also look in `/opt/gurobi` and `/opt/gurobi560`. On a Mac system, it will also look in `/Library/gurobi` and `/Library/gurobi560`. 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:\gurobi560`.

The `grbgetkey` program will suggest a location in which to store your license key file, but it also allows you to choose an alternate location. If you accept its suggestion, there is nothing else to do. Otherwise, please read on.

While we allow you to store your license key file in a non-default location, we'd like to strongly recommend that you use the default location, particularly if you are settting up a token server or a compute server. Setting up a non-default location is quite error-prone and a frequent source of trouble.

If you would still like to use a non-default license key file location, you can do so by setting environment variable `GRB_LICENSE_FILE` to point to the license key file. *Important note: the environment variable should point to the license key file itself, not to the directory that contains the file.*

Setting an environment variable - Windows

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.*

Setting an environment variable - Linux

On Linux systems, if you choose to put the license key file in a non-default location, you should add a line like the following to your `.bashrc` file:

```
export GRB_LICENSE_FILE=/usr/home/jones/gurobi.lic
```

For Linux `csh` shell users, you should add the following to your `.cshrc` file:

```
setenv GRB_LICENSE_FILE /usr/home/jones/gurobi.lic
```

You should of course set the variable to point to the actual location of your license key file.

Setting an environment variable - Mac

On Mac systems, you can set the optional `GRB_LICENSE_FILE` environment variable through `environment.plist`, as explained in Apple's Runtime configuration Guidelines.

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. If you get an error message that indicates that your hostname is not recognized as belonging to an academic domain, you should continue reading this section...

No reverse DNS information

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

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

...it means that 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*, to be more specific) for the DHCP device itself.

Note that 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 for more information about reverse DNS.

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 acad
```

...it means that your domain isn't on our academic domain list. Please make sure you are connected to your university network. If the reported host name is a valid university address, please send the specific error message you receive to `support@gurobi.com` and we'll add your domain.

Connecting to your academic domain through a VPN will typically not present a problem for validation, since the request will appear to originate from inside your academic domain. However, some VPNs are configured to use *split tunneling*, where traffic to public internet sites is routed through your ISP. Split tunneling introduces a major security hole into the private network, so it is not that common. If you find that your validation request is rejected when you are connected to your VPN, 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. Unfortunately, such configurations are incompatible with our academic validation process.

3.1 Setting up a token server

Important note: most Gurobi licenses do not use the token server. You should only follow these instructions if your `gurobi.lic` file contains the line `TYPE=TOKEN`.

If you are using a floating or single-use license, you will need to choose a machine to act as the Gurobi token server before you can use the Gurobi Optimizer. 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. For a single-use license, the client and the token server must be the same machine. For a floating-license, 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.

Your next step depends on the type of machine you will be using as your token server. Instructions for Windows and Linux token servers follow. Note that a Mac OS system cannot currently be used as a token server.

Token server clients also require licenses, but you can generate the required license files yourself. Consult the section on client licenses for details.

Setting up a Windows 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.

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. You do this by selecting *Allow a program through Windows firewall* under the *Security* area of the Control Panel (labeled *Allow an app through Windows firewall* in Windows 8). You should add `grb_ts.exe` to the list of exceptions.

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 so far, 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 (also known as a `cmd` window) that has administrator privileges. Running `grb_ts -h` lists command-line options. Issuing a `grb_ts -s` command stops a running license service. Issuing a `grb_ts -v` command starts 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.

To upgrade 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.

Windows services can be stubborn. If the new token server refuses to start, you may need to delete the old one manually. To do so, type `sc delete grb_ts` from a `cmd` window that has administrator privileges.

All output from the Gurobi Token Server goes to the Windows Event Log. You can access this in Windows Vista or Windows 7 through the Event Viewer. Type *Event* in the search box under the *Start* menu to launch the viewer.

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.

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` from a `cmd` window.

Setting up a Linux token server

On a Linux system, you will need to start the token server daemon by running program `grb_ts` (with no arguments) on your token server machine. You only need to do this once — the token server will keep running until you stop it (or until the machine is shut down). Be sure that the license key file has been installed before running this program. Note that the token server runs as a user process, so you do not need root privileges to start it.

Note that if you would like the token server to restart when the machine is rebooted, you should ask your system administrator to start it from `/etc/rc.local`. If your Gurobi installation and license key file are in their default locations, then adding the following should suffice:

`/opt/gurobi560/linux64/bin/grb_ts`

To stop a running token server, you can issue the `grb_ts -s` command. You can also use the `ps` command to find the relevant process ID, and the Linux `kill` command to terminate that process.

To upgrade 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.

Output from the token server goes to the system log (`/var/log/syslog`). You should see a message similar to the following when you start the server:

`Mar  9 12:37:21 mymachine grb[7917]: Gurobi Token Server started: Sat Mar  9 12:37:21 2013`

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.

Firewalls

If you run into trouble accessing the token server, check to see if the server machine is running firewall software (like *Bastille* or *ipfilter*) that is blocking access to some ports. The Gurobi token server uses port number 41954 by default, so you'll need to open access to that port on the server. Please consult the documentation for your firewall software to determine how to do this. If there's a conflict on the default port, you can choose a different one by adding a `PORT` line to both the server and the client license key files:

```
PORT=46325
```

You can choose any available port number.

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.

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`.

3.2   Setting up a compute server

Important note: most Gurobi licenses do not use the compute server. You should only follow these instructions if your `gurobi.lic` file contains the line `CSENABLED=1`.

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).

We should note that our distributed parallel tuning feature allows any machine to be configured as a *restricted compute server*. These special Gurobi compute servers don't require a Gurobi license. The only jobs they will accept are tuning jobs. To set up a restricted compute server, you should follow the instructions for setting up a standard compute server, but you can ignore any step that involves a license file.

Your next step depends on the types of machines you will be using as your compute servers. Instructions are available for Windows, Linux, or Mac OS.

Compute server clients also require licenses, but you can generate the required license files yourself. Consult the client license section for details.

Setting up a Windows compute server

On a Windows system, you will need to start the Compute Server service by selecting the *Gurobi Compute 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.

Firewalls

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

Some machines have more restrictive firewalls that may require additional action. The Gurobi compute server uses ports 61000-65000 by default. If you are unable to reach the compute server after taking the steps described so far, you should ask your network administrator for more information on how to open the required ports.

Compute server parameters

Note that a compute server has a few user-configurable parameters. You can set these by creating a `grb_cs.cnf` file and placing it in the same directory as `grb_cs.exe`. Please consult the *Gurobi Compute Server* section of the Gurobi Reference Manual for details.

Starting and stopping the grb_cs Gurobi service

Once the compute server service has been started, you should see the `grb_cs` 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 Compute Server* item on this screen.

You can also start or stop the Gurobi Compute Server service from a console window (also known as a `cmd` window) that has administrator privileges. Running `grb_cs -h` lists command-line options. Issuing a `grb_cs -s` command stops a running compute server. Issuing a `grb_cs -v` command starts the compute server in verbose mode. Verbose mode produces a log message (in the Windows Event Log) each time a client starts a job.

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

1. Stop the old compute 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 compute server.

Windows services can be stubborn. If the new compute server refuses to start, you may need to delete the old one manually. To do so, type `sc delete grb_cs` from a `cmd` window that has administrator privileges.

All output from the Gurobi Compute Server goes to the Windows Event Log. You can access this in Windows Vista or Windows 7 through the Event Viewer. Type *Event* in the search box under the *Start* menu to launch the viewer.

Next steps

Clients of a compute server require simple license files. Your next step is to set up a client license.

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

You can test the state of the compute server at any time, as well as get a list of both running and queued client jobs, by typing `gurobi_cl ---clients` from a `cmd` window.

Setting up a Linux compute server

On a Linux system, you will need to start the Compute Server daemon by running program `grb_cs` (with no arguments) on your compute server machine. You only need to do this once — the compute server will keep running until you stop it (or until the machine is shut down). Be sure that the license key file has been installed before running this program. Note that the compute server runs as a user process, so you do not need root privileges to start it.

Note that if you would like the compute server to restart when the machine is rebooted, you should ask your system administrator to start it from `/etc/rc.local`. If your Gurobi installation and license key file are in their default locations, then adding the following should suffice:

`/opt/gurobi560/linux64/bin/grb_cs`

Compute server parameters

Note that a compute server has a few user-configurable parameters. You can set these by creating a `grb_cs.cnf` file and placing it in the same directory as `grb_cs`. Please consult the *Gurobi Compute Server* section of the Gurobi Reference Manual for details.

Starting and stopping the grb_cs Gurobi daemon

To stop a running compute server, you can issue the `grb_cs -s` command. You can also use the `ps` command to find the relevant process ID, and the Linux `kill` command to terminate that process.

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

1. Stop the old compute 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 compute server.

Output from the compute server goes to the system log (`/var/log/syslog`). You should see a message similar to the following when you start the server:

`Mar  9 12:37:21 mymachine grb[7917]: Gurobi Compute Server started: Sat Mar  9 12:37:21 2013`

By default, the compute server only produces logging output when it starts. To obtain more detailed logging information, start the compute server with the `-v` switch. This will produce a log message each time a client starts a job.

Firewalls

If you run into trouble accessing the compute server, check to see if the server machine is running firewall software (like *Bastille* or *ipfilter*) that is blocking access to some ports. The Gurobi compute server uses port numbers 61000-65000 by default, so you'll need to open access to these ports on the server. Please consult the documentation for your firewall software to determine how to do this. If there's a conflict on the default port, you can choose a different one by adding a `PORT` line *to both the server and the client license key files*:

```
PORT=46325
```

You can choose any available port number.

Next steps

Clients of a compute server sometimes require simple license files. Your next step is to set up a client license.

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

You can test the state of the compute server at any time, as well as get a list of both running and queued client jobs, by typing `gurobi_cl ---clients`.

Setting up a Mac OS compute server

On a Mac system, you will need to start the Compute Server daemon by running program `grb_cs` (with no arguments) on your compute server machine. You only need to do this once — the compute server will keep running until you stop it (or until the machine is shut down). Be sure that the license key file has been installed before running this program. Note that the compute server runs as a user process, so you do not need root privileges to start it.

Compute server parameters

Note that a compute server has a few user-configurable parameters. You can set these by creating a `grb_cs.cnf` file and placing it in the same directory as `grb_cs`. Please consult the *Gurobi Compute Server* section of the Gurobi Reference Manual for details.

Starting and stopping the grb_cs Gurobi daemon

To stop a running compute server, you can issue the `grb_cs -s` command. You can also use the `ps` command to find the relevant process ID, and the Mac OS `kill` command to terminate that process.

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

1. Stop the old compute 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 compute server.

Output from the compute server goes to the system log (`/var/log/system.log`). You will need to modify `/etc/syslog.conf` to see these messages, since by default OS X only allows error message in the system log. Once you have modified `syslog.conf`, you should see a message similar to the following when you start the server:

```
Mar  9 12:37:21 mymachine grb[7917]: Gurobi Compute Server started: Sat Mar  9 12:37:21 2013
```

By default, the compute server only produces logging output when it starts. To obtain more detailed logging information, start the compute server with the `-v` switch. This will produce a log message each time a client starts a job.

Firewalls

If you run into trouble accessing the compute server, check to see if the server machine is running firewall software that is blocking access to some ports. The Gurobi compute server uses port numbers 61000-65000 by default, so you'll need to open access to these ports on the server. Please consult the documentation for your firewall software to determine how to do this. If there's a conflict on the default port, you can choose a different one by adding a `PORT` line *to both the server and the client license key files*:

```
PORT=46325
```

You can choose any available port number.

Next steps

Clients of a compute server sometimes require simple license files. Your next step is to set up a client license.

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

You can test the state of the compute server at any time, as well as get a list of both running and queued client jobs, by typing `gurobi_cl ---clients`.

3.3 Creating a client license

As noted earlier, some types of Gurobi licenses require you to set up both a server license and client licenses. We've already covered the steps involved in setting up a token server or a compute server. This section is devoted to setting up the client side.

For both token servers and compute servers, the purpose of a client license is quite simple: it tells the client where to find the Gurobi server(s). The details of the client licenses differ slightly, depending on the server type...

Client for a token server

For a token server, the client `gurobi.lic` file should contain a line that looks like this:

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

or:

```
TOKENSERVER=192.168.1.100
```

This line gives the name or IP address of the token server.

Note that if your client and server are both running on the same machine, you have two options. The first is to add a `TOKENSERVER=localhost` line to your `gurobi.lic` file. The token server will ignore this line, and the client will ignore everything but this line. Your second option 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.

Client for a compute server

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 API's, these API's provide routines that allow you to specify the names of the compute servers. If you use these routines, then Gurobi licenses aren't required on the client.

Alternatively, 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. The client `gurobi.lic` file should contain a line that looks like this:

`COMPUTESERVER=machine1.mydomain.com,machine2.mydomain.com,machine3.mydomain.com`

or like this:

`COMPUTESERVER=192.168.1.100,192.168.1.101,192,168.1.102`

This line provides a comma-separated list of Gurobi compute servers. If your compute servers use a password, you should also include a line that gives the password:

`PASSWORD=cspwd`

If you'd like to specify a job priority, you can add a line that gives an integer priority value:

`PRIORITY=7`

Higher priority jobs run before lower priority jobs. Please consult the *Gurobi Compute Server* section of the Gurobi Reference Manual for more information.

Note that if your client and server are both running on the same machine, you can add a `COMPUTESERVER=localhost` line to your `gurobi.lic` file. The compute server will ignore this line, and the client will ignore everything but this line. Another option in this situation 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.

Client license file location

We recommend that you place your client `gurobi.lic` file in the default location for your platform:

- Windows: `c:\gurobi`

- Linux: `/opt/gurobi`

- Mac: `/Library/gurobi`

You can store the license file in other locations. Read this section to learn more about using a non-default license file 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 setting up a token server or setting up a compute server for more information.

Testing the license

Once you have obtained a license key for your machine, you are ready to test your license using the Gurobi Interactive Shell. On Windows systems, double-click on the Gurobi icon on your desktop. On Linux or Mac OS systems, type `gurobi.sh` in a *Terminal* window. The shell should produce the following output:

```
Gurobi Interactive Shell, Version 5.6.0
Copyright (c) 2013, Gurobi Optimization, Inc.
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 that looks like the following:

```
Server capacity available on myserver - running now
```

If you see similar output, 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. Read this section for more information on where Gurobi looks for this file. Note that on Windows systems, this error is often caused by a hidden file suffix. Make sure the name of your license file is `gurobi.lic`, and not `gurobi.lic.txt`.

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 compute server, the following message...

```
ERROR: No response from servers
```

indicates that the compute server isn't currently running. Please consult the section on setting up a compute server.

Some Windows users have reported that they were unable to launch the Gurobi shell after running the installer. You may need to log off and log back in again in order for the environment variable changes made by the installer to take effect.

Solving a Simple Model – The Gurobi Command Line

We'll now present a simple math programming model, show you how that model would be captured in a file, and then show you 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.

The problem statement – producing coins

We 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 that are used to produce the coins that are put into circulation. Imagine that the Mint 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

Let's imagine that 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. Then we'll define a linear objective function. This is the function we'd like to minimize (or maximize). Finally, 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 wide 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 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 5.6.0 in the recommended location, you'll find the file here:

- Windows (64-bit): `c:\gurobi560\win64\examples\data\coins.lp`

- Linux: `/opt/gurobi560/linux64/examples/data/coins.lp`

- Mac OS: `/Library/gurobi560/mac64/examples/data/coins.lp`

Feel free to open the file in a text editor. However, before you consider making any modifications to this file or creating your own, we should point out a few rules about LP format files. One relates to the ordering of the various 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.

The second rule is that 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`.

The third important rule is that variables 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
```

Another important property of LP files is that variables have 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.

As we mentioned earlier, 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 here. As we've mentioned, our command-line interface 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 bring up a window that allows you to run command-line programs. On a Linux or Mac system, you can use a *Terminal* window. On a Windows system, you'll need to bring up a `console` window (also known as a `cmd` window). To open a console window, press the *Start* and *R* keys on your keyboard simultaneously, and then type `cmd` into the *Run* window that pops up. Alternatively, you can type `cmd` into the *Search* box

that appears in the bottom-left after clicking on the Windows *Start* button. (Note that the Gurobi Interactive Shell, which was used earlier to test your license, does *not* directly accept command-line program input, so it is not an appropriate choice for this section).

The name of the Gurobi command-line tool is `gurobi_cl`. To invoke it, we simply need to type `gurobi_cl`, followed by the name of the model file. For example, if our example is stored in file `c:\gurobi560\win64\examples\data\coins.lp`, you would type the following command into your command-line window...

```
> gurobi_cl c:\gurobi560\win64\examples\data\coins.lp
```

This command should produce the following output...

```
Read LP format model from file c:\gurobi560\win64\examples\data\coins.lp
Reading time = 0.00 seconds
(null): 4 rows, 9 columns, 16 nonzeros
Optimize a model with 4 rows, 9 columns and 16 nonzeros
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)
Found heuristic solution: objective 26.1000000
Found heuristic solution: objective 113.3000000

Root relaxation: objective 1.134615e+02, 4 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  113.30000  113.46154  0.14%     -    0s
     0     0  113.45952    0    1  113.30000  113.45952  0.14%     -    0s
H    0     0                       113.4500000  113.45952  0.01%     -    0s

Explored 0 nodes (5 simplex iterations) in 0.00 seconds
Thread count was 2 (of 2 available processors)

Optimal solution found (tolerance 1.00e-04)
Best objective 1.134500000000e+02, best bound 1.134500000000e+02, gap 0.0%
```

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. If we request a `.sol` file...

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

...then the command produces 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 of course 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 PyScripter IDE for Windows. In our opinion, PyScripter strikes a nice balance between power and simplicity. If you are a Windows user and would prefer to use a graphical environment over a more command-line driven environment, we suggest that you install PyScripter now. You can also consult the PyScripter instructions for pointers to other IDE options that might be of interest if you are on a different platform or would like to try a different Windows Python IDE.

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:

- 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`.

From Linux or Mac OS, you can simply type `gurobi.sh` from the command prompt. 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 (or gurobi.sh for Linux or Mac OS)

Gurobi Interactive Shell, Version 5.6.0
Copyright (c) 2013, Gurobi Optimization, Inc.
Type "help()" for help

gurobi> m = read('c:/gurobi560/win64/examples/data/coins.lp')
Read LP format model from file c:/gurobi560/win64/examples/data/coins.lp
Reading time = 0.00 seconds
(null): 4 rows, 9 columns, 16 nonzeros
gurobi> m.optimize()
Optimize a model with 4 rows, 9 columns and 16 nonzeros
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)
Found heuristic solution: objective 26.1000000
Found heuristic solution: objective 113.3000000

Root relaxation: objective 1.134615e+02, 4 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  113.30000  113.46154  0.14%     -    0s
     0     0  113.45952    0     1  113.30000  113.45952  0.14%     -    0s
 H   0     0                         113.4500000  113.45952  0.01%     -    0s

Explored 0 nodes (5 simplex iterations) in 0.00 seconds
Thread count was 4 (of 4 available processors)

Optimal solution found (tolerance 1.00e-04)
Best objective 1.134500000000e+02, best bound 1.134500000000e+02, gap 0.0%
```

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:/gurobi560/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()
Optimize a model with 4 rows, 9 columns and 16 nonzeros
Presolve removed 2 rows and 5 columns
Presolve time: 0.00s
Presolved: 2 rows, 4 columns, 5 nonzeros

MIP start did not produce a feasible 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                  0    71.9000000   71.90000   0.0%     -    0s

Explored 0 nodes (2 simplex iterations) in 0.00 seconds
Thread count was 4 (of 4 available processors)

Optimal solution found (tolerance 1.00e-04)
Best objective 7.190000000000e+01, best bound 7.190000000000e+01, gap 0.0%
```

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:/gurobi560/win64/examples/data/glass4')
Read MPS format model from file c:/gurobi560/win64/examples/data/glass4.mps
Reading time = 0.00 seconds
glass4: 396 Rows, 322 Columns, 1815 NonZeros
gurobi> m.optimize()
```

```
Optimize a model with 396 Rows, 322 Columns and 1815 NonZeros
Presolve removed 4 rows and 5 columns
Presolve time: 0.00s
Presolved: 392 Rows, 317 Columns, 1815 Nonzeros
Found heuristic solution: objective 3.691696e+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 3.6917e+09 8.0000e+08  78.3%     -    0s
     0     0 8.0000e+08     0    72 3.6917e+09 8.0000e+08  78.3%     -    0s
     0     0 8.0000e+08     0    72 3.6917e+09 8.0000e+08  78.3%     -    0s
     0     0 8.0000e+08     0    72 3.6917e+09 8.0000e+08  78.3%     -    0s
     0     2 8.0000e+08     0    72 3.6917e+09 8.0000e+08  78.3%     -    0s
H  769   732                      2.800024e+09 8.0000e+08  71.4%   5.2   0s
H  834   781                      2.666693e+09 8.0000e+08  70.0%   5.3   0s
H 1091   984                      2.475023e+09 8.0000e+08  67.7%   5.1   0s
H 1092   986                      2.400020e+09 8.0000e+08  66.7%   5.1   0s
H 1092   984                      2.380021e+09 8.0000e+08  66.4%   5.1   0s
H 1095   988                      2.350020e+09 8.0000e+08  66.0%   5.1   0s
* 1845  1543            94         2.316685e+09 8.0000e+08  65.5%   4.9   0s
* 2131  1627           126         2.150018e+09 8.0000e+08  62.8%   4.8   0s
H 2244  1580                      2.100019e+09 8.0000e+08  61.9%   4.8   0s
H 2248  1341                      1.900018e+09 8.0000e+08  57.9%   5.0   0s
H 3345  1816                      1.900018e+09 8.0000e+08  57.9%   4.1   0s
H 3346  1744                      1.900017e+09 8.0000e+08  57.9%   4.1   0s
H15979 10383                      1.900017e+09 8.0000e+08  57.9%   2.5   1s
H19540 13051                      1.900016e+09 8.0000e+08  57.9%   2.4   1s
*21124 13489           101         1.866683e+09 8.0000e+08  57.1%   2.4   1s
*23011 14690           100         1.850015e+09 8.0000e+08  56.8%   2.3   1s
*25630 15679           143         1.800016e+09 8.0000e+08  55.6%   2.3   1s
*28365 15421           113         1.700015e+09 8.0000e+08  52.9%   2.3   1s
H29910 16333                      1.700014e+09 8.0000e+08  52.9%   2.3   1s
*30582 16765           124         1.700014e+09 8.0000e+08  52.9%   2.3   1s
*33238 16251            92         1.677794e+09 8.0000e+08  52.3%   2.3   1s
*37319 18258            85         1.633349e+09 8.0000e+08  51.0%   2.2   1s
H40623 19584                      1.600015e+09 8.0000e+08  50.0%   2.3   2s
 81781 42951 1.1000e+09    49    51 1.6000e+09 8.0001e+08  50.0%   2.2   5s
 199990 100088 1.6000e+09    82    28 1.6000e+09 8.0001e+08  50.0%   2.3  10s
*242810 116891            97         1.600015e+09 8.2001e+08  48.8%   2.3  11s
*243703 116786            95         1.600014e+09 8.2001e+08  48.8%   2.3  11s

Interrupt request received
```

```
Explored 255558 nodes (588336 simplex iterations) in 12.36 seconds
Thread count was 8 (of 8 available processors)

Solve interrupted
Best objective 1.6000142000e+09, best bound 8.5000490000e+08, gap 46.8752%
```

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()
gurobi> m.optimize()
Optimize a model with 396 Rows, 322 Columns and 1815 NonZeros
Presolve removed 4 rows and 5 columns
Presolve time: 0.00s
Presolved: 392 Rows, 317 Columns, 1815 Nonzeros
Found heuristic solution: objective 3.691696e+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 3.6917e+09 8.0000e+08   78.3%      -    0s
      0       0 8.0000e+08      0   72 3.6917e+09 8.0000e+08   78.3%      -    0s
      0       0 8.0000e+08      0   72 3.6917e+09 8.0000e+08   78.3%      -    0s
      0       0 8.0000e+08      0   73 3.6917e+09 8.0000e+08   78.3%      -    0s
H     0       0                     3.075022e+09 8.0000e+08   74.0%      -    0s
H     0       0                     3.020023e+09 8.0000e+08   73.5%      -    0s
      0       0 8.0000e+08      0   76 3.0200e+09 8.0000e+08   73.5%      -    0s
      0       0 8.0000e+08      0   75 3.0200e+09 8.0000e+08   73.5%      -    0s
H     0       0                     2.550024e+09 8.0000e+08   68.6%      -    0s
H     0       2                     2.175020e+09 8.0000e+08   63.2%      -    0s
      0       2 8.0000e+08      0   75 2.1750e+09 8.0000e+08   63.2%      -    0s
H    95      98                     2.150020e+09 8.0000e+08   62.8%    4.6    0s
H    96      98                     2.120018e+09 8.0000e+08   62.3%    4.6    0s
H   101     103                     2.116687e+09 8.0000e+08   62.2%    4.5    0s
H   110     103                     2.100017e+09 8.0000e+08   61.9%    4.3    0s
H   352     325                     2.000018e+09 8.0000e+08   60.0%    4.2    0s
H   406     375                     1.991686e+09 8.0000e+08   59.8%    4.0    0s
H  1074     888                     1.981836e+09 8.0000e+08   59.6%    3.5    0s
H  1078     889                     1.966686e+09 8.0000e+08   59.3%    3.5    0s
H  1107     878                     1.900018e+09 8.0000e+08   57.9%    3.5    0s
H  1696    1125                     1.800017e+09 8.0000e+08   55.6%    3.4    0s
H  1845    1146                     1.800017e+09 8.0000e+08   55.6%    4.2    1s
H  1863    1087                     1.733350e+09 8.0000e+08   53.8%    4.3    1s
H  2353    1273                     1.733350e+09 8.0000e+08   53.8%    4.3    1s
H  2517    1299                     1.700016e+09 8.0000e+08   52.9%    4.3    1s
H  2598    1248                     1.666682e+09 8.0000e+08   52.0%    4.3    1s
H  2733    1252                     1.633349e+09 8.0000e+08   51.0%    4.2    1s
  14259    7927 1.5000e+09     85   28 1.6333e+09 8.0000e+08   51.0%    3.5    5s
  24846   14278 1.1000e+09     49   55 1.6333e+09 8.0001e+08   51.0%    3.5   10s
H25035   13985                     1.600016e+09 8.0001e+08   50.0%    3.5   10s
H25066   14020                     1.600016e+09 8.0001e+08   50.0%    3.5   10s
H25072   13532                     1.583350e+09 8.0001e+08   49.5%    3.5   10s
H26218   14083                     1.575016e+09 8.0001e+08   49.2%    3.5   10s
H26326   14118                     1.566682e+09 8.0001e+08   48.9%    3.5   10s
H26577   13650                     1.525016e+09 8.0001e+08   47.5%    3.5   10s
```

Interrupt request received

Cutting planes:
  Gomory: 6
  Implied bound: 26
  MIR: 60

Explored 30546 nodes (107810 simplex iterations) in 11.81 seconds
Thread count was 8 (of 8 available processors)

```
Solve interrupted
Best objective 1.5250155750e+09, best bound 8.0000520000e+08, gap 47.5412%
```

Results are consistent with our expectations. We find a better solution sooner by shifting the focus towards finding feasible solutions (objective value `1.525e9` versus `1.6e9`).

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 12 parameter sets in 47.77s

Baseline parameter set: runtime 2.39s

Improved parameter set 1 (runtime 1.72s):

     RINS 0
```

In this case, it found that setting the `RINS` parameter to 0 for model `misc07` reduced the runtime from 2.39s to 1.72s.

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, particulary if your model has severe numerical issues.

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

```
> grbtune c:\gurobi560\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
```

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 ignored.

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:/gurobi560/win64/examples/data/p0033')
Read MPS format model from file c:/gurobi560/win64/examples/data/p0033.mps
Reading time = 0.00 seconds
P0033: 16 Rows, 33 Columns, 98 NonZeros.
gurobi> b = read('c:/gurobi560/win64/examples/data/stein9')
Read MPS format model from file c:/gurobi560/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>
b : <gurobi.Model MIP instance STEIN9: 13 constrs, 9 vars>
```

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:

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

in order to import this function. 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. On Linux or Mac OS, you will need to open a terminal window, change your current directory to the Gurobi `<installdir>` (the directory that contains the file `setup.py`), and issue the following command:

```
python setup.py install
```

Unless you are using your own private Python installation, you will need to run this command as super-user. Once `gurobipy` is successfully installed, you can type `import gurobipy` or `from gurobipy import *` from your Python shell and access all of the Gurobi classes and methods.

Note that for this installation to succeed, your Python environment must be compatible with the Gurobi Python module. You should only install 32-bit Gurobi libraries into a 32-bit Python shell (similarly for 64-bit versions). In addition, your Python version must be compatible. With this release, `gurobipy` can be used with Python 2.7 or 3.2 on Windows and Linux, and with Python 2.7 on Mac OS.

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);
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:

$$
\begin{array}{rrcrcrcrcr}
\textbf{maximize} & x & + & y & + & 2\,z & & \\
\textbf{subject to} & x & + & 2\,y & + & 3\,z & \leq & 4 \\
& x & + & y & & & \geq & 1 \\
& \multicolumn{7}{c}{x,\ y,\ z \text{ binary}}
\end{array}
$$

Example mip1_c.c

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

```c
#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;
  int       zero = 0;

  /* Create environment */

  error = GRBloadenv(&env, "mip1.log");
```

```c
if (error || env == NULL) {
  fprintf(stderr, "Error: could not create environment\n");
  exit(1);
}

/* 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;

/* Integrate new variables */

error = GRBupdatemodel(model);
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, NULL);
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, NULL);
if (error) goto QUIT;
```

```c
  /* 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:

```
  error = GRBloadenv(&env, "mip1.log");
  if (error || env == NULL) {
    fprintf(stderr, "Error: could not create environment\n");
    exit(1);
  }
```

Later attempts to create optimization models will always require an environment, so environment creation should always be the first step when using the Gurobi optimizer. The second argument to *GRBloadenv()* provides the name of the Gurobi log file. If the argument is an empty string or NULL, no log file will be written.

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);
```

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;
```

Updating the model – lazy modification

Model modifications in the Gurobi optimizer are done in a *lazy* fashion, meaning that the effects of the modifications are not seen immediately. This approach makes it easier to perform a sequence of model modifications, since the model doesn't change while it is being modified. However, lazy modifications do require you to manually integrate model changes when needed. This is done with the following routine:

```
/* Integrate new variables */
error = GRBupdatemodel(model);
if (error) goto QUIT;
```

In our example, the model would contain zero variables immediately before the call to *GRBupdatemodel()*, and three immediately after. Later attempts to add constraints to the model without first making this call would fail, since the model would contain no variables.

Adding constraints to the model

Once the new variables are integrated into the model, the next step is to add our two constraints. 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, NULL);
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:

```
/* 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, NULL);
if (error) goto QUIT;
```

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:

```
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...

```
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 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 = GRBgetdoubleattr(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 = GRBgetdoublearrayattr(model, GRB_DBL_ATTR_X, 0, 3, x);
if (error) goto QUIT;
printf("  x=%.0f, y=%.0f, z=%.0f", x[0], x[1], x[2]);
```

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:

```
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.

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_2008.sln`, `C_examples_2010.sln`, and `C_examples_2012.sln` (Visual Studio 2008, 2010, and 2012 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. For Linux or Mac OS platforms, the `<installdir>/examples/build` directory contains an example Makefile. Typing `make mip1_c` will build and run this 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:

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

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`)...

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

int
main(int    argc,
     char *argv[])
{
  try {
    GRBEnv env = GRBEnv();

    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");

    // Integrate new variables

    model.update();
```

```
  // 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):

```
GRBEnv env = GRBEnv();
```

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:

```
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");
GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB_BINARY, "z");
```

Variables are added through the `addVar()` method on the model object. 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.

Updating the model - lazy modification

Model modifications in the Gurobi optimizer are done in a *lazy* fashion, meaning that the effects of the modifications are not seen immediately. This approach makes it easier to perform a sequence of model modifications, since the model doesn't change while it is being modified. However, lazy modifications do require you to manually integrate model changes when needed. This is done with the *update* method:

```
// Integrate new variables
model.update();
```

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 expression 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 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 easy to build linear 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;
cout << y.get(GRB_StringAttr_VarName) << " "
     << y.get(GRB_DoubleAttr_X) << endl;
cout << z.get(GRB_StringAttr_VarName) << " "
     << z.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.

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_2008.sln`, `C++_examples_2010.sln`, and `C++_examples_2012.sln` (Visual Studio 2008, 2010, and 2012 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. For Linux or Mac OS platforms, the `<installdir>/examples/build` directory contains an example Makefile. Typing `make mip1_c++` will build and run this 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:

- On Windows, 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++md2010.lib` when you choose runtime library option /MD in Visual Studio 2010. Similarly, use file `gurobi_c++mtd2012.lib` when you choose runtime library option /MTd in Visual Studio 2012.

- A C++ program that uses Gurobi must link in both the Gurobi C++ library (e.g., `gurobi_c++mt2010.lib` on Windows, `libgurobi_c++.a` on Linux and Mac) *and* the Gurobi C library (`gurobi56.lib` on Windows, `libgurobi56.so` on Linux and Mac).

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).

Our example optimizes the following model:

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

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`)...

```
import gurobi.*;

public class Mip1 {
  public static void main(String[] args) {
    try {
      GRBEnv    env   = new GRBEnv("mip1.log");
      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");

      // Integrate new variables

      model.update();

      // 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):

```
GRBEnv env = new GRBEnv("mip1.log");
```

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:

```
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");
GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "z");
```

Variables are added through the `addVar()` method on a model object. 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.

Updating the model - lazy modification

Model modifications in the Gurobi optimizer are done in a *lazy* fashion, meaning that the effects of the modifications are not seen immediately. This approach makes it easier to perform a sequence of model modifications, since the model doesn't change while it is being modified. However, lazy modifications do require you to manually integrate model changes when needed. This is done with the *update* method:

```
// Integrate new variables
model.update();
```

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 constraints. The first constraint is added here:

```
// Add constraint: x + 2 y + 3 z <= 4
GRBLinExpr expr;

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));
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));
```

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:

```
model.dispose();
env.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.

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. For Linux or Mac OS platforms, the `<installdir>/examples/build` directory contains an example Makefile. Typing `make Mip1` will build and 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.

This section 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. 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).

Our example optimizes the following model:

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

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`)...

```
using System;
using Gurobi;

class mip1_cs
{
  static void Main()
  {
    try {
      GRBEnv    env   = new GRBEnv("mip1.log");
      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");

      // Integrate new variables

      model.Update();

      // 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.Get(GRB.StringAttr.VarName)
                                + " " + x.Get(GRB.DoubleAttr.X));
        Console.WriteLine(y.Get(GRB.StringAttr.VarName)
                                + " " + y.Get(GRB.DoubleAttr.X));
        Console.WriteLine(z.Get(GRB.StringAttr.VarName)
                                + " " + z.Get(GRB.DoubleAttr.X));

        Console.WriteLine("Obj: " + model.Get(GRB.DoubleAttr.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):

```
GRBEnv env = new GRBEnv("mip1.log");
```

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:

```
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");
GRBVar z = model.AddVar(0.0, 1.0, 0.0, GRB.BINARY, "z");
```

Variables are added through the `AddVar()` method on a model object. 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.

Updating the model - lazy modification

Model modifications in the Gurobi optimizer are done in a *lazy* fashion, meaning that the effects of the modifications are not seen immediately. This approach makes it easier to perform a sequence of model modifications, since the model doesn't change while it is being modified. However, lazy modifications do require you to manually integrate model changes when needed. This is done with the *Update* method:

```
// Integrate new variables
model.Update();
```

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 expression 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 constraints:

```
// 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");
```

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 easy 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, x);
expr.AddTerm(3.0, x);
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. In particular, we can query the `VarName` and `X` attributes to obtain the name and solution value for each variable:

```
Console.WriteLine(x.Get(GRB.StringAttr.VarName) + " " + x.Get(GRB.DoubleAttr.X));
Console.WriteLine(y.Get(GRB.StringAttr.VarName) + " " + y.Get(GRB.DoubleAttr.X));
Console.WriteLine(z.Get(GRB.StringAttr.VarName) + " " + z.Get(GRB.DoubleAttr.X));
```

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

```
Console.WriteLine("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:

```
model.Dispose();
env.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.

Building and running the example

You can use the `CS_examples_2008.sln`, `CS_examples_2010.sln`, or `CS_examples_2012.sln` solution files in `<installdir>/examples/build` to build and run the example with Visual Studio 2008, 2010, or 2012, respectively. Clicking on the `mip1_cs` project, and then selecting *Run* from the *Build* menu will compile and run 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.

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. Finally, our Python interface 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. We've already introduced the Interactive Shell in an earlier section. This section will work through two 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 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 online Python tutorial.

Note that Gurobi does not require a separate Python installation; the Gurobi distribution includes all the tools needed to run Python programs. You will need to use a set of scripts we provide in order to run Gurobi Python programs within the Python environment we distribute. Alternatively, if you are already a Python user, we provide tools for installing the `gurobipy` module in 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 PyScripter IDE for Windows. In our opinion, PyScripter strikes a nice balance between power and simplicity. If you are a Windows user and would prefer to use a graphical environment over a more command-line driven environment, we suggest that you install PyScripter now. You can also consult the PyScripter instructions for pointers to other IDE options that might be of interest if you are on a different platform or would like to try a different Windows Python IDE.

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.

11.1   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:

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

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`)...

```python
from gurobipy import *

try:

    # Create a new model
    m = 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")

    # Integrate new variables
    m.update()

    # 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")

    m.optimize()

    for v in m.getVars():
        print v.varName, v.x

    print 'Obj:', m.objVal

except GurobiError:
    print 'Error reported'
```

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:

```
from gurobipy import *
```

Gurobi Python applications should always start with this line.

Note that in order for this command to succeed, the Python application needs to know how to find the Gurobi functions and classes. Recall that you have two 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` (Windows) or `gurobi.sh mip1.py` (Linux and Mac). The second option 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:

```
m = 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")
y = m.addVar(vtype=GRB.BINARY, name="y")
z = m.addVar(vtype=GRB.BINARY, name="z")
```

Variables are added through the `addVar()` method on a model object. 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()`.

Updating the model - lazy modification

Model modifications in the Gurobi optimizer are done in a *lazy* fashion, meaning that the effects of the modifications are not seen immediately. This approach makes it easier to perform a sequence of model modifications, since the model doesn't change while it is being modified. However, lazy modifications do require you to manually integrate model changes when needed. This is done with the *update* method:

```
# Integrate new variables
m.update()
```

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 Python API overloads the arithmetic operators to allow you to build linear and quadratic expression 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 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 easy 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 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:', 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.

Running the example

When you run the example (`gurobi.bat mip1.py` on Windows, or `gurobi.sh mip1.py` on Linux or Mac), you should see the following output:

```
Optimize a model with 2 rows, 3 columns and 5 nonzeros
Presolve removed 2 rows and 3 columns
Presolve time: 0.00s

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

Optimal solution found (tolerance 1.00e-04)
Best objective 3.000000000000e+00, best bound 3.000000000000e+00, gap 0.0%
x 1.0
y 0.0
z 1.0
Obj: 3.0
```

11.2 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 constructs: `lists`, `tuples`, `dictionaries`, `list comprehension`, and the `tuplelist` class. The first four are standard Python concepts that are particularly important in our interface, while the last is a custom class 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 simply ordered collections of Python objects. A list is created and displayed 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

List comprehension is an important Python feature that allows you to build lists 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> print [x*x for x in [1, 2, 3, 4, 5]]
[1, 4, 9, 16, 25]
```

A list comprehension can contain more than one `for` clause, and it can contain 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 3 and are not equal:

```
gurobi> print [(x,y) for x in range(3) for y in range(3) if x != y]
[(0, 1), (0, 2), (1, 0), (1, 2) (2, 0), (2, 1)]
```

(Details on the `range` function can be found here). List comprehension is used extensively in our Python examples.

The tuplelist class

The final 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 specified 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 use a '*' string to indicate that any value is acceptable in that position in the tuple.

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

```
gurobi> print l.select(1, '*')
[(1, 2), (1, 3)]
gurobi> print l.select('*', 3)
[(1, 3), (2, 3)]
gurobi> print l.select(1, 3)
[(1, 3)]
gurobi> print l.select('*', '*')
[(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, '*')
[(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
[(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)]
```

Returning to our network flow example, once we've built a `tuplelist` containing all valid commodity-source-destination combinations on the network (we'll call it `flows`), we can select all arcs that flow into a specific destination city as follows:

```
gurobi> inbound = flows.select('*', '*', '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)...

```
from gurobipy import *

# Model data

commodities = ['Pencils', 'Pens']
nodes = ['Detroit', 'Denver', 'Boston', 'New York', 'Seattle']

arcs, capacity = multidict({
  ('Detroit', 'Boston'):   100,
  ('Detroit', 'New York'):  80,
  ('Detroit', 'Seattle'):  120,
  ('Denver',  'Boston'):   120,
  ('Denver',  'New York'): 120,
  ('Denver',  'Seattle'):  120 })
arcs = tuplelist(arcs)

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 }

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 = Model('netflow')

# Create variables
flow = {}
for h in commodities:
    for i,j in arcs:
        flow[h,i,j] = m.addVar(ub=capacity[i,j], obj=cost[h,i,j],
                               name='flow_%s_%s_%s' % (h, i, j))
m.update()

# Arc capacity constraints
for i,j in arcs:
    m.addConstr(quicksum(flow[h,i,j] for h in commodities) <= capacity[i,j],
                'cap_%s_%s' % (i, j))

# Flow conservation constraints
for h in commodities:
    for j in nodes:
        m.addConstr(
          quicksum(flow[h,i,j] for i,j in arcs.select('*',j)) +
              inflow[h,j] ==
          quicksum(flow[h,j,k] for j,k in arcs.select(j,'*')),
                  'node_%s_%s' % (h, j))

# Compute optimal solution
m.optimize()

# Print solution
if m.status == GRB.status.OPTIMAL:
    for h in commodities:
        print '\nOptimal flows for', h, ':'
        for i,j in arcs:
            if flow[h,i,j].x > 0:
                print i, '->', j, ':', flow[h,i,j].x
```

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:

```
from gurobipy import *
```

We then create a few lists that contain model data:

```
commodities = ['Pencils', 'Pens']
nodes = ['Detroit', 'Denver', 'Boston', 'New York', 'Seattle']

arcs, capacity = multidict({
  ('Detroit', 'Boston'):   100,
  ('Detroit', 'New York'):  80,
  ('Detroit', 'Seattle'):  120,
  ('Denver',  'Boston'):   120,
  ('Denver',  'New York'): 120,
  ('Denver',  'Seattle'):  120 })
arcs = tuplelist(arcs)
```

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`).

In our example, we plan to use `arcs` to select subsets of the arcs when building constraints later. We therefore pass the `list` of tuples returned by `multidict` to the `tuplelist` constructor to create a `tuplelist` object instead.

The model also requires cost data for each commodity-arc pair:

```
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:

```
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 stored in a dictionary `flow`:

```
flow = {}
for h in commodities:
  for i,j in arcs:
    flow[h,i,j] = m.addVar(ub=capacity[i,j], cost=cost[h,i,j],
                           name='flow_%s_%s_%s' % (h, i, j))
m.update()
```

The `flow` variable is triply subscripted: by commodity, source node, and destination node. Note that the dictionary 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:

```
for i,j in arcs:
  m.addConstr(quicksum(flow[h,i,j] for h in commodities) <= capacity[i,j],
              'cap_%s_%s' % (i, j))
```

Note that we use list comprehension to build a list of all variables associated with an arc `(i,j)`:

```
flow[h,i,j] for h in commodities
```

(To be precise, as we've used it here, this is actually called a *generator expression* in Python, but it is similar enough to list comprehension that you can safely ignore the difference for the purpose of understanding this example). The result is passed into the `quicksum` function to create a Gurobi linear expression that captures the sum of all of these variables. The Gurobi `quicksum` function is an alternative to the Python `sum` function that is much faster for building large expressions.

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:

```
for h in commodities:
  for j in nodes:
    m.addConstr(
      quicksum(flow[h,i,j] for i,j in arcs.select('*',j)) + inflow[h,j] ==
      quicksum(flow[h,j,k] for j,k in arcs.select(j,'*')),
               'node_%s_%s' % (h, j))
```

Results

Once we've added the model constraints, we call `optimize` and then output the optimal solution:

```
if m.status == GRB.status.OPTIMAL:
  for h in commodities:
    print '\nOptimal flows for', h, ':'
    for i,j in arcs:
      if flow[h,i,j].x > 0:
        print i, '->', j, ':', flow[h,i,j].x
```

If you run the example (`gurobi.bat netflow.py` on Windows, or `gurobi.sh netflow.py` on Linux and Mac), you should see the following output:

```
Optimize a model with 16 rows, 12 columns and 36 nonzeros
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   0.000000e+00      0s

Solved in 0 iterations and 0.00 seconds
Optimal objective  5.500000000e+03

Optimal flows for Pencils :
Detroit -> Boston : 50.0
Denver -> New York : 50.0
Denver -> Seattle : 10.0

Optimal flows for Pens :
Detroit -> Boston : 30.0
Detroit -> New York : 30.0
Denver -> Boston : 10.0
Denver -> Seattle : 30.0
```

11.3   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
```

For Linux or Mac OS platforms, type:

```
gurobi.sh mip1.py
```

If you are a Python user, and wish to use Gurobi from within your own Python environment, you can install the `gurobipy` module directly into your environment. 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. On Linux or Mac OS, you will need to open a terminal window, change your current directory to the Gurobi `<installdir>` (the directory that contains the file `setup.py`), and issue the following command:

```
python setup.py install
```

Unless you are using your own private Python installation, you will need to run this command as super-user. Once `gurobipy` is successfully installed, you can type `python mip1.py` (more generally, you can type `from gurobipy import *` in your Python environment).

MATLAB Interface

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 (e.g., `/opt/gurobi560/linux64/matlab` for the 64-bit Linux version of Gurobi 5.6). To get started, type the following commands within MATLAB to change to the `matlab` directory and call `gurobi_setup`:

```
>> cd /opt/gurobi560/linux64/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. For example, if you are using the 64-bit version of MATLAB, you'll need to install the 64-bit version of Gurobi, and you'll need to use the 64-bit Gurobi MATLAB libraries (i.e., the ones included with the 64-bit version of 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:

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

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`)...

```
names = {'x'; 'y'; 'z'};

try
    clear model;
    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';

    clear params;
    params.outputflag = 0;
    params.resultfile = 'mip1.lp';

    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);

catch gurobiError
    fprintf('Error reported\n');
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. A few fields are mandatory: the constraint matrix (`A`), the objective vector (`obj`), the
right-hand side vector (`rhs`), and the constraint sense vector (`sense`). A model can also include
optional fields (e.g., the objective sense `modelsense`).

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.

Subsequent statements populate other fields of the `model` variable, including the objective vector,
the right-hand-side vector, and the constraint sense vector.

In addition to the mandatory fields, this example also sets two optional 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 fields 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, a `LP` format file that contains the optimization model). The Gurobi MATLAB interface allows you to set as many Gurobi parameters as you would 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 value. 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 (e.g., `/opt/gurobi560/linux64/examples/matlab` for the 64-bit Linux version of Gurobi 5.6). 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 /opt/gurobi560/linux64/examples/matlab
>> mip1
```

If Gurobi was successfully set up for use in MATLAB, you should see the following output:

```
       status: 'OPTIMAL'
  versioninfo: [1x1 struct]
       objval: 3
      runtime: 0.0386
            x: [3x1 double]
        slack: [2x1 double]
     objbound: 3
    itercount: 0
 baritercount: 0
    nodecount: 0


x 1
y 0
z 1
Obj: 3.000000e+00
```

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

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>')
```

The Gurobi R package file can be found in the `<installdir>/R` directory of your Gurobi installation (e.g., `/opt/gurobi560/linux64/R` for the 64-bit Linux version of Gurobi 5.6). You should browse the `<installdir>/R` directory to find the exact name of the file for your platform (the 64-bit Linux package is in file `gurobi_5.6-0_R_x86_64-pc-linux-gnu.tar.gz`, while the Windows package is in file `gurobi_5.6-0.zip`).

You will need to be careful that the R binary and the Gurobi package you install both use the same instruction set. For example, if you are using the 64-bit version of R, you'll need to install the 64-bit version of Gurobi, and the 64-bit Gurobi R package. 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:

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

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`)...

```
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)
```

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. A few components are mandatory: the constraint matrix (`A`), the objective vector
(`obj`), the right-hand side vector (`rhs`), and the constraint sense vector (`sense`). 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 components 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`.

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 recommand Model Building in Mathematical Programming by H.P. Williams.

While command-line tools are likely to be familiar to Linux users and to most Mac users, we realize that the command line can be quite foreign to a Windows user. This section guides you through the steps involved in installing PyScripter, a free and widely-used Python Integrated Development Environment (IDE) for Windows. PyScripter makes it easier for Windows users to use the Gurobi Interactive Shell, and to develop and debug programs that use the Gurobi Python interface.

If you are using Gurobi on a Linux or Mac platform, or if you would like to consider other IDE options on Windows, a number of other free Python IDEs are available. Popular choices include Eric, iep, and PyDev. We won't be covering the details of installing these other options for use with Gurobi, but the PyScripter instructions that follow should provide a good outline for the steps involved. We've found that PyScripter provides a nice balance between power and complexity, but we realize that people may look for different things in their IDEs.

Step 1: Install Gurobi

The first step in using Gurobi from Pyscripter is to install Gurobi on your machine and install a Gurobi license (if you haven't already done so).

Step 2: Install Python

The next step is to install a stand-alone Python interpreter. Be sure to install a version that is compatible with the version of Gurobi you installed. We recommend Python 2.7, but Python 3.2 is also an option:

Choose the `Windows X86-64 Installer` if you installed the 64-bit version of Gurobi, or the `Windows Installer` if you installed the 32-bit version of Gurobi.

Note that Python gives you the option of choosing an install directory, but PyScripter will only be able to find it if you stick with the default.

Step 3: Install Gurobi into Python

The third step is to run a simple program that installs the Gurobi module into Python. Simply double-click on `pysetup` in the `bin` folder of your Gurobi installation (`c:\gurobi560\win64\bin` for a default installation of the 64-bit Windows version). The program will prompt you for the location of your Python installation. When the program is finished, you should see output that looks like the following:

### Step 4: Install PyScripter

The final step is to install PyScripter:

Again, choose the `x64-Setup` version if you installed the 64-bit version of Gurobi, or the `Setup` version if you installed the 32-bit version of Gurobi.

One installation of PyScripter can work with multiple versions of Python. You will need to launch the one that corresponds to the Python version you installed (we used `PyScripter for Python 2.7` in this example):
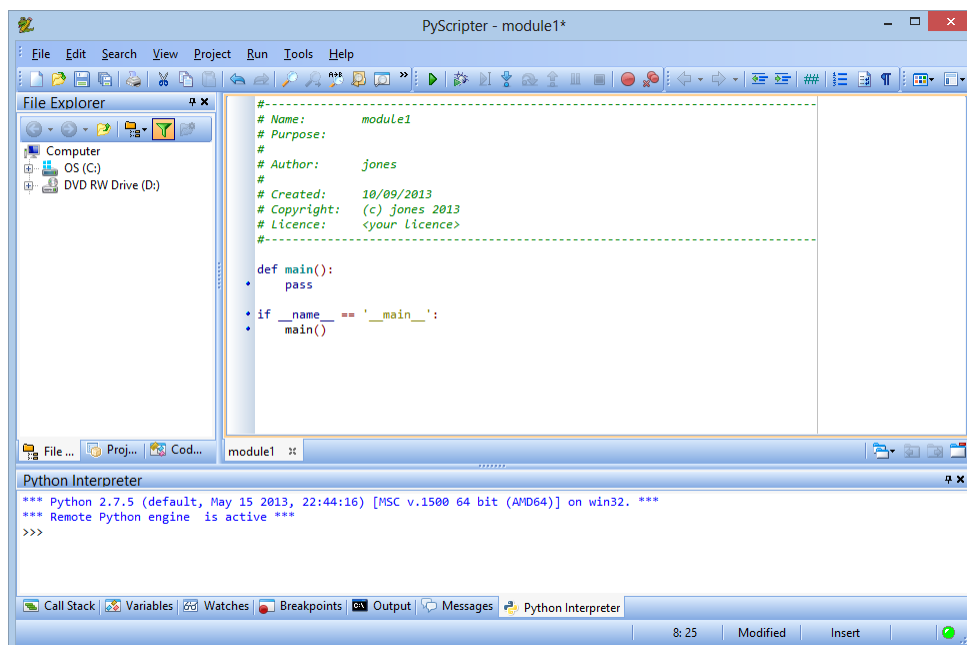
Using PyScripter

Gurobi Interactive Shell commands can be typed directly into the `Python Interpreter` window of PyScripter:
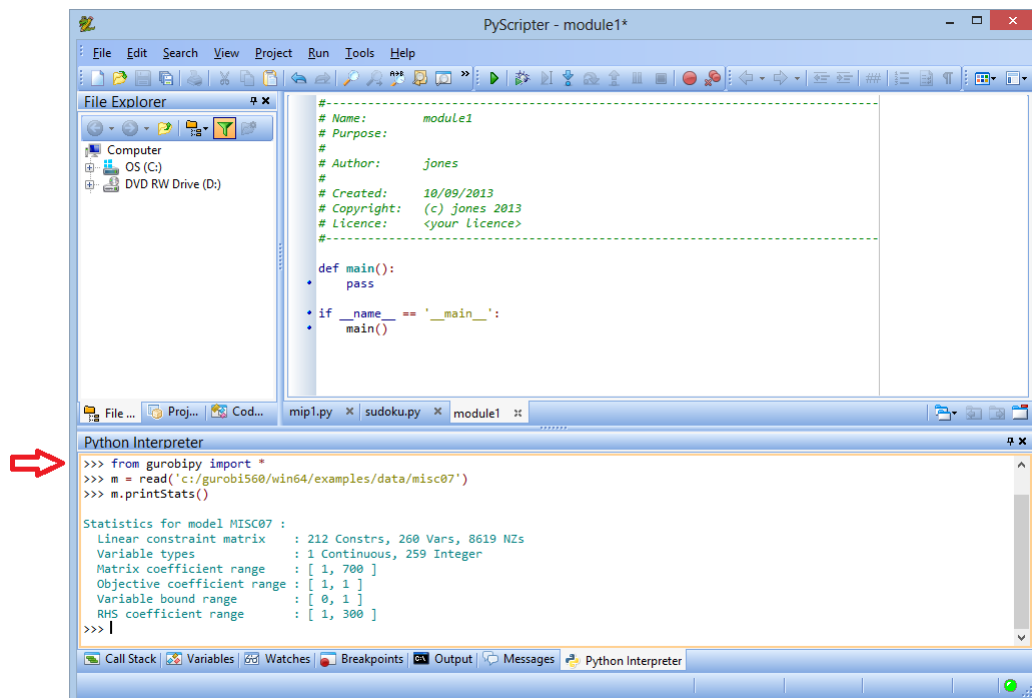

Unfortunately, a general-purpose Python IDE like PyScripter requires one extra step that isn't required when you launch the Gurobi shell from the Gurobi icon or by using the `gurobi.sh` command: you must type `from gurobipy import *` before issuing any Gurobi commands. Unlike our icon or our `gurobi.sh` command, the IDE won't load the Gurobi module automatically, so you must do it manually.

You can also use PyScripter 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, you should see:
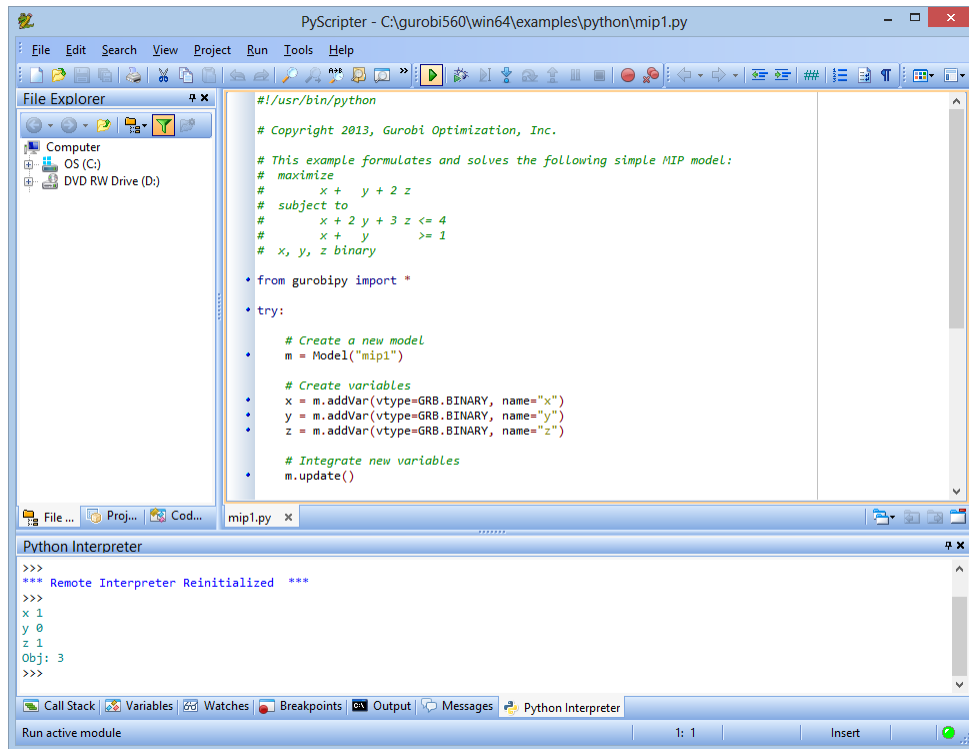
Some Gurobi examples require command-line arguments. Those can be input from the `Command Line Parameters...` item of the `Run` menu. For example, to run the `sudoku.py` example with file `sudoku1` as input...
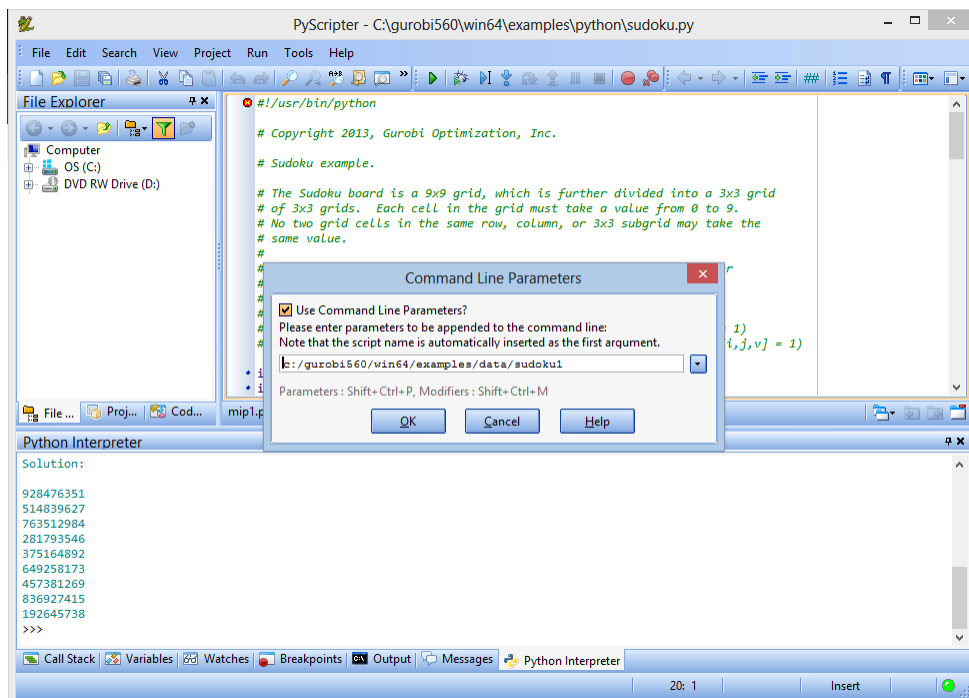
A few Gurobi examples require additional Python modules. For example, our `diet4` example uses the Python xlrd module to extract data from an Excel spreadsheet. You can find missing Python modules at the Python Package Index (PyPI) site. The Python interpreter that we include with our distribution includes all the modules used by our examples. When you install your own Python interpreter, you may have to install some modules yourself from this site.

This section briefly describes the purposes of the more important files in the Gurobi distribution. The Windows, Linux, and Mac OS distributions have mostly the same file structure, but some files differ between them, so we present them separately.

Note that the lists 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 is installed.

Windows file organization

The following files and directories are created in your installation directory (`c:\gurobi560\win32` by default for the 32-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

  - Gurobi56.NET.XML - Visual Studio help for .NET wrapper
  - Gurobi56.NET.dll - .NET wrapper
  - GurobiJni56.dll - Java JNI wrapper
  - grb_cs.exe - Gurobi Compute Server executable
  - grb_csw.exe - Gurobi Compute Server executable
  - 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
  - gurobi56.dll - Gurobi native DLL (used by all Gurobi interfaces)
  - 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.pdf - Quick Start guide - PDF
  - refman - Reference Manual - HTML (open index.html in this directory)
  - refman.pdf - Reference Manual - PDF

- examples

  - build - Visual Studio projects for C, C++, C#, and Visual Basic examples; run*.bat files for 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
  - 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

- include

  - gurobi_c++.h - C++ include file
  - gurobi_c.h - C include file

- lib

  - gurobi.jar - Java interface
  - gurobi.py - Python startup file
  - gurobi56.lib - Gurobi library import file
  - gurobi_c++md2008.lib - C++ interface (when using -MD compiler switch with Visual Studio 2008)
  - gurobi_c++md2010.lib - C++ interface (when using -MD compiler switch with Visual Studio 2010)
  - gurobi_c++md2012.lib - C++ interface (when using -MD compiler switch with Visual Studio 2012)
  - gurobi_c++mdd2008.lib - C++ interface (when using -MDd compiler switch with Visual Studio 2008)
  - gurobi_c++mdd2010.lib - C++ interface (when using -MDd compiler switch with Visual Studio 2010)
  - gurobi_c++mdd2012.lib - C++ interface (when using -MDd compiler switch with Visual Studio 2012)

- gurobi_c++mt2008.lib - C++ interface (when using -MT compiler switch with Visual Studio 2008)

- gurobi_c++mt2010.lib - C++ interface (when using -MT compiler switch with Visual Studio 2010)

- gurobi_c++mt2012.lib - C++ interface (when using -MT compiler switch with Visual Studio 2012)

- gurobi_c++mtd2008.lib - C++ interface (when using -MTd compiler switch with Visual Studio 2008)

- gurobi_c++mtd2010.lib - C++ interface (when using -MTd compiler switch with Visual Studio 2010)

- gurobi_c++mtd2012.lib - C++ interface (when using -MTd compiler switch with Visual Studio 2012)

- matlab - Gurobi MATLAB interface

- python27 - Python 2.7 files used by the interactive shell and the Python interface (no need to look inside this directory)

- python32 - Python 3.2 files (no need to look inside this directory)

- R - R Gurobi package

- setup.py - Python setup file - for installing the gurobipy module into your own Python environment

Linux file organization

The following files and directories are created in your installation directory
(typically `/opt/gurobi560/linux64` for the 64-bit Linux distribution):

- EULA.pdf - Gurobi End User License Agreement - PDF format

- ReleaseNotes.html - release notes

- bin

  - grb_cs - Gurobi Compute Server executable
  - grb_csw - Gurobi Compute Server executable
  - grb_ts - Gurobi Token Server executable
  - grbgetkey - retrieves your Gurobi license key from the Gurobi key server
  - grbprobe - probes system details (typically not used)
  - grbtune - parameter tuning tool
  - gurobi_cl - simple command-line binary
  - gurobi.env - sample parameter initialization file
  - gurobi.sh - starts the Gurobi interactive shell

- python2.7 - Python shell

- 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.pdf - Quick Start guide - PDF
  - refman - Reference Manual - HTML (open index.html in this directory)
  - refman.pdf - Reference Manual - PDF

- examples

  - build - Makefile for C, C++, Java, and Python examples
  - c - source code for C examples
  - c# - source code for C# examples (for Windows)
  - c++ - source code for C++ examples
  - data - data files for 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
  - python2.7 - Dummy Python include files (no need to look inside this directory)

- lib

  - gurobi.jar - Java interface
  - gurobi.py - Python startup file
  - libgurobi56.so - Gurobi library (symbolic link to current version)
  - libgurobi_c++.a - C++ interface (symbolic link)
  - libgurobi_g++4.1.a - C++ interface (when using g++ 4.1 - e.g., on a Red Hat 5 system)
  - libgurobi_g++4.2.a - C++ interface (when using g++ 4.2 or later)
  - libGurobiJni56.so - Java JNI wrapper
  - libgurobi.so.5.6.0 - Gurobi native library (used by all interfaces)
  - python2.7 - Python files used by the interactive shell and the Python interface (no need to look inside this directory)

- python2.7_utf16 - Python 2.7 files for use with UTF-16 Python versions (no need to look inside this directory)

  - python2.7_utf32 - Python 2.7 files for use with UTF-32 Python versions (no need to look inside this directory)

  - python3.2_utf16 - Python 3.2 files for use with UTF-16 Python versions (no need to look inside this directory)

  - python3.2_utf32 - Python 3.2 files for use with UTF-32 Python versions (no need to look inside this directory)

- matlab - Gurobi MATLAB interface

- R - R Gurobi package

- setup.py - Python setup file - for installing the gurobipy module into your own Python environment

Mac OS file organization

The following files and directories are created in your installation directory
(typically `/Library/gurobi560/mac64`):

- EULA.pdf - Gurobi End User License Agreement - PDF format

- ReleaseNotes.html - release notes

- bin

  - grb_cs - Gurobi Compute Server executable
  - grb_csw - Gurobi Compute Server executable
  - grb_ts - Gurobi Token Server executable
  - grbgetkey - retrieves your Gurobi license key from the Gurobi key server
  - grbprobe - probes system details (typically not used)
  - grbtune - parameter tuning tool
  - gurobi.env - sample parameter initialization file
  - gurobi.sh - starts the Gurobi interactive shell
  - gurobi_cl - 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.pdf - Quick Start guide - PDF
  - refman - Reference Manual - HTML (open index.html in this directory)

- refman.pdf - Reference Manual - PDF

- examples

  - build - Makefile for C, C++, Java, and Python examples
  - c - source code for C examples
  - c# - source code for C# examples (for Windows)
  - c++ - source code for C++ examples
  - data - data files for 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.py - Python startup file
  - gurobipy - Python files used by the interactive shell and the Python interface (no need to look inside this directory)
  - libGurobiJni56.jnilib - Java JNI wrapper
  - libgurobi56.so - Gurobi native library (used by all interfaces)
  - libgurobi_c++.a - C++ interface (symbolic link)
  - libgurobi_g++4.2.a - C++ interface

- matlab - Gurobi MATLAB interface

- R - R Gurobi package

- setup.py - Python setup file - used by the installer to install the gurobipy module into your Python environment