University of Colorado Denver

Department of Mathematical and Statistical Sciences

University Of Colorado Denver

Master's Project

A Tutorial on Optimization Solvers in Different Programming Languages

Kushmakar Baral

Advisor

Dr. Steffen Borgwardt

October 2, 2019

Contents

1 Abstract

Learning a new programming language to solve a mathematical program can be challenging. For many practitioners, it would be easiest to be able to access optimization solvers in one of the programming languages with which they are familiar.

This project provides detailed instructions on the installation and use of optimization solvers such as CPLEX, GUROBI, and XPRESS, as well as information on accessing these solvers in five different programming languages. The core programming languages used are AMPL, MATLAB, R, Python, and C++. We show sample codes in these programming languages for linear, integer, quadratic, nonlinear, and semi-definite programs.

2 Optimization Solvers and Installation

This section describes the optimization solvers and instructions on installation processes of the optimization solvers: CPLEX, GUROBI, XPRESS, and IPOPT.

2.1 CPLEX

CPLEX is an optimization software package. The CPLEX Optimizer offers flexible, high-functioning mathematical programming solvers for linear programming, mixed-integer programming, quadratic programming, and quadratically constrained programming problems. It can be used in several programming languages such as C++, C, Python, MATLAB, etc. There is a form that needs to be filled out and registered to obtain a free version of IBM CPLEX OPTIMIZER, which can be found on this link:

`https://www.ibm.com/account/reg/us-en/signup?formid=urx-20028`

Once the form is completed, users will be given an option to download it.

2.2 GUROBI

GUROBI is a commercial optimization solver for linear programming, quadratic programming, quadratically constrained programming, mixed-integer linear programming, mixed-integer quadratic programming, and mixed-integer quadratically constrained programming. GUROBI was founded in 2008 and is named after its founders, Zonghao *Gu*, Edward *Ro*thberg and Robert *Bi*xby. It can support various programming and modeling languages such as

C++, Java, Python, C, MATLAB, R, AMPL, etc. First, one has to download the Gurobi Optimizer and obtain the license key to use the optimizer. There are three options for downloading the latest version of GUROBI. In the first option, "Gurobi Optimizer", comes with interfaces for C++, C, MATLAB, and Python. In the second option, "Gurobi Solver for AMPL", comes with the solver Gurobi, and this is what one needs if they have AMPL installed in their computer. In the third option, "AMPL and Gurobi Software", comes with both AMPL software and Gurobi Solver; this is what one needs if both AMPL software and Gurobi Solver is not installed in their computer. Once the Gurobi solver is downloaded, then there will be a need for a license key to be able to activate the solver. To obtain the academic license the user needs to request using their own school's information and must get their computer connected to the school's internet server. Then, Gurobi Optimization will grant a license key which can be used to activate the solver. Finally, Gurobi can be used to solve mathematical problems. The link to download Gurobi and obtain the license is `https://www.gurobi.com/downloads/`.

2.3 XPRESS

XPRESS is an optimization solver for linear programming, mixed-integer programming, convex quadratic programming, quadratically constrained programming problem, and second-order cone programming. XPRESS has the interfaces for C, C++, Java, Python, AMPL, GAMS, MATLAB, etc. To download and install the XPRESS solver, one will have to fill out a form at the end of the page from the link:

```
https://content.fico.com/xpress-optimization-community-license
```

Once installation is completed, one will get an option to use community license, which is free, or one can buy the other options. After the activation with the community license, one can use the XPRESS solver.

2.4 IPOPT

IPOPT is an open-source interior-point optimizer which can solve various optimization problems ranging from linear programming to nonlinear programming. It is a part of COIR-OR project. It is written in Fortran and C, so it is not very easy to implement it in other programming languages. However, for the purpose here, this will be used to solve the nonlinear programming problem in Python. Since it is an open-source optimizer, we install in the Python by typing this in the Conda PowerShell prompt:

```
conda install -c conda-forge ipopt
```

To download and install this solver in other programming languages is plenty of practical work. Hence, we will skip it for convenience.

2.5 Other Optimization Solvers

CBC

It is an open-source linear programming and mixed-integer programming solver. It is a default solver in PuLP library in python.

CLP

It is an open-source linear programming solver written in C++. It can be a bit slow but is designed to solve linear programming problems with up to millions of variables and constraints. It uses the simplex algorithm as the main algorithm.

Minos

MINOS is a Fortran software package for solving linear and nonlinear mathematical programs. It is supported and compatible in the AIMMS, AMPL, APMonitor, GAMS, and TOMLAB modeling systems. It is not compatible with programming languages such as MATLAB, Python, and R.

MOSEK

Mosek is an optimization solver for linear, mixed-integer, quadratic, quadratically constraint, conic and convex nonlinear mathematical programs. It can also solve semidefinite programming and second-order cone programming problems. However, MOSEK is well known in the financial industry for its state-of-the-art optimizers for quadratic and conic problems (MOSEK.com).

SCIP

SCIP is one of the non-commercial solvers for mixed-integer programming and mixed-integer nonlinear programming (Scip.zib.de). It has several built-in packages to solve mathematical programs. However, the most attractive package that it has that most of the other solvers do not have is the SDP-package which can be used to solve semidefinite programming problems. There are specific installation steps in the following link : `http://www.opt.tu-darmstadt.de/~smars/SDP/scip_sdp_preprint.pdf`

GLPK

GLPK is a solver which can solve large-scale linear programming and mixed-integer programming problems. However, it cannot solve anything higher than an integer programming problem. To install this in python, type the following in the Conda PowerShell prompt:

```
conda install -c conda-forge glpk
```

To download and install this in other programming languages is much more than just typing a command. Hence for the sake of convenience, we will leave it at this.

3 Types of Mathematical Programs

3.1 Linear Programming

A linear program (LP) is an optimization problem in which the objective function is linear in the unknowns and the constraints consist of linear equalities and linear inequalities (Luenberger & Ye, 1984). The exact form of the constraints may differ from problem to problem, but any of the Linear programs can be transformed into standard form:

$$\begin{aligned} \textbf{min } & c^T x \\ \textbf{subject to } & Ax = b \\ & \mathbf{x} \geq 0 \end{aligned}$$

Here $\mathbf{x}$ is an n-dimensional column vector, $\mathbf{c}^T$ is an n-dimensional row vector, $\mathbf{A}$ is an mxn matrix, and $\mathbf{b}$ is an m-dimensional column vector. The vector inequality $\mathbf{x} \geq 0$ means that each component of $\mathbf{x}$ is nonnegative.

3.2 Integer Programming

Integer programming is a solution method for discrete optimization problems. The class of optimization problem with objective function and constraints, where constraints can only be integers. The general mathematical formula of integer programming is as follows:

$$\begin{aligned} \textbf{min } & c^T x \\ \textbf{subject to } & Ax = b \; or Ax \leq b \\ & x \geq 0, x \in \mathbb{Z}^+ \end{aligned}$$

3.3 Quadratic Programming

A quadratic programming problem is a class of nonlinear optimization problem in which the objective function is quadratic and all the constraints are linear. The general mathematical formula of Quadratic programming is as follows:

$$\begin{aligned} \textbf{max or min } & \frac{1}{2} c^T x + x^T Q x \\ \textbf{subject to } & Ax = b \; or Ax \leq b \\ & x \geq 0 \end{aligned}$$

where Q is symmetric positive semi-definite matrix, $c, x \in \mathbb{R}^n$, $b \in \mathbb{R}^n$ and A is a mxn matrix.

3.4 Nonlinear Programming

A nonlinear program, like a linear program, has continuous variables; the expressions in its objective and constraints need not be linear, but they must represent "smooth" functions (Fourer, R. Gay, D.& Kernighan, B.,1987) The nonlinear optimization problems are divided into two classes: unconstrained problems and constrained problems. Unconstrained problems have no equality or inequality constraints are present. The generic form of unconstrained problem is:

$$\begin{aligned} \textbf{min } & f(x) \\ \textbf{subject to } & x \in \mathbb{R}^n \end{aligned}$$

where f is a real-valued function and $\mathbb{R}^n$ is the feasible set.

Constrained problems involves at least one inequality or equality constraint. The generic form of constrained problem is:

$$\begin{aligned} \textbf{min } & f(x) \\ \textbf{subject to } & h_i(x) = 0, g_j(x) \leq 0 \\ & x \in \mathbb{R}^n \end{aligned}$$

where $m \leq n$ and the functions $f, h_i, i = 1, 2, ..., m$ and $g_j, j = 1, 2, ..., p$ are continuous, and usually assumed to possess continuous second partial derivatives.

3.5 Semidefinite Programming

The semidefinite programming problem is essentially an ordinary linear program where the nonnegativity constraint is replaced by a semidefinite constraint on matrix variable. The standard form for the primal problem is:

$$\begin{aligned} \textbf{min } & CX \\ \textbf{subject to } & A_k \cdot X = b_k, k = 1, ..., m \\ & X \succeq 0 \end{aligned}$$

where A_k, C, X are in space S^n where S^n is the space of $n \times n$ symmetric matrices, and b_k is a scalar, and the constraint $X \succeq 0$ means that, X, the unknown variable, must lie in the convex cone of positive semidefinite matrices.

4 Solvers in different programming languages

The mathematical programming problems from Appendix 6.1 will solved in five programming languages using the solvers: CPLEX, GUROBI, XPRESS, and IPOPT.

4.1 Python

It is essential to let Python know that the solvers are in the computer. CPLEX and GUROBI solver package will come with python folder which is needed to let python know that the solver is in the computer. In the python folder of the solver packages, there will be a file named "setup.py".

So, it is necessary for us to install that file in the command prompt or conda powershell prompt after accessing the correct directory of the "setup.py" file in the solver package. We use following command to install the setup.py file:

```
python setup.py install
```

However, for XPRESS, we can install it from PyPI or Conda repository by using the following commands respectively:

```
pip install xpress
conda install -c fico-xpress xpress
```

Once we let python know that all the solvers are in the system, we can use the solvers. To use the optimization solvers, it is vital to install some of the libraries in python which can call the solvers. So, the user can install PuLP library which can call various optimization solvers such as COIN, CPLEX, GLPK, GUROBI, XPRESS, etc for linear and integer programming problems. Hence, to install PuLP, it is necessary to run either of code block below in the Jypter notebook or the Conda Powershell Prompt, respectively.

```
pip install pulp
conda install -c conda-forge pulp
```

What is PuLP?

Pulp is a free software written in python which describes optimization problems as mathematical models. It can call several external linear programming solvers such as CPLEX, GLPK, CBC, GUROBI, etc. to solve those models, then it uses python commands to manipulate and display the solution. However, the default solver in Pulp is CBC from COIN-OR, which is good enough for simple linear programming optimization problems.

PuLP is a good python library which can be used to solve linear and integer programming problems. However, PuLP does not support quadratic or nonlinear programming problems. Therefore, a library named Pyomo is essential to solving quadratic, and nonlinear programming problems. Installing this library in python requires one to use either of the following commands:

```
pip install pyomo

conda install -c conda-forge pyomo
```

Now, pyomo can be used to solve quadratic, and nonlinear programming problems. Pyomo library supports most of the commercial and non-commercial optimization solvers. But, the solvers that are discussed here are CPLEX, GUROBI, XPRESS, and IPOPT, which it supports.

Once Pyomo is installed, it can call the solvers CPLEX, GUROBI, XPRESS, and IPOPT to solve a simple Linear, Integer, Quadratic, and nonlinear programming problems.

Calling CPLEX, GUROBI, XPRESS, and IPOPT in Python
Once the PuLP library is installed in python, it is easy to call CPLEX and solve linear programming problem from equation (1) in Appendix 6.1. The associated python script is in Appendix 6.2.1. To solve using solver GUROBI, the user has to call GUROBI as the solver. Following is the only line of the sample code that needs to be changed in the program in Appendix 6.2.1 for calling GUROBI:

```
status = prob.solve(GUROBI())
```

Similarly, to solve using solver XPRESS, call XPRESS as the solver, to do that, one has to type XPRESS in place of the solver. Following is the only line of the sample code that needs to be changed in the program in Appendix 6.2.1:

```
status = prob.solve(XPRESS())
```

However, no matter which solver is used the optimal solution for the LP from equation (1) is

```
objective: 133.333
x1 = 2.667
x2 = 2.667
```

In order to solve integer programming problem from equation (2) in Appendix 6.1, the code will be slightly changed. Instead of variables as continuous, we restrict those variable to integers. All the sample code from Appendix 6.2.1 stays same, except two lines below:

```
x1 = LpVariable ("x1", lowBound=0, cat='Integer')
x2 = LpVariable ("x2", lowBound=0, cat='Integer')
```

Here the variables are categorized as integers to make the linear programming problem an integer programming problem. Then, the solving process is exactly the same as the linear programming problem. The associated python script is in Appendix 6.2.2. If one needs to change the solver around, then one can make the change accordingly. However, no matter what solver is used, the solution for equation (2) is:

```
(3.0, 2.0, 130.0)
```

Now, to solve quadratic programming problem from equation (3), the Pyomo library should be used because PuLP can only deal with linear and integer programming problems. First of all, when writing the script to solve quadratic programming problem, the necessary modules needed from the library should be imported, to do that the following command is needed in the script:

```
from pyomo.environ import *
```

Then, create the model using the ConcreteModel() function from the module. The necessary objective function and constraints will be used to make the complete model. Next, the solver will be called to solve the quadratic programming problem from equation (3). Any of the three solvers CPLEX, GUROBI, and XPRESS can be used to solve the quadratic problem. The associated python script for the quadratic programming problem from equation (3) is in Appendix 6.2.3. To change the solver around, the following code from Appendix 6.2.3 can be changed for the specific solver.

```
solver = SolverFactory('cplex')
```

But, no matter which solver is used, the solution should be:

```
Status = optimal

x1 = 0.500000

x2 = 1.000000

Objective = 11.250000
```

The CPLEX, GUROBI, and XPRESS cannot solve nonlinear programming problems. Thus, a different solver is needed to solve the nonlinear programming problem from equation (4). Since IPOPT is an open-source nonlinear solver, we can use it to solve the nonlinear programming problem. To do so, first, the user will have to install the IPOPT solver using the following command in the Conda PowerShell prompt.

```
conda install -c conda-forge ipopt
```

This will install the IPOPT solver in python, then Pyomo library will be able to use it to solve the problem. The associated python script is in Appendix 6.2.4. The solution for the nonlinear programming problem from equation (4) is:

```
Status = optimal
x1 = 0.000000
x2 = 3.000000
Objective = -6.000000
```

The optimal objective is negative in the output because the Pyomo library solves the problem as a minimization problem, so it is necessary to negate the objective function to make it a maximization problem. Thus, negating the optimal objective is necessary to get the correct objective. Hence, the optimal objective is positive 6 not negative 6.

Finally, there are not very many solvers to do semi-definite programming. However, python has a pre-built package called CVXOPT, which can be installed to solve semi-definite programming problems. So, to install CVXOPT, the following command should be used in conda powershell prompt

```
conda install -c conda-forge cvxopt
```

Then, necessary modules should be imported to model and solve the problem. The following command is the first line of the python script to solve the semi-definite programming problem from equation (5).

```
from cvxopt import matrix, solvers
```

Then, the sdp solver is called to solve the problem. The associated python script for solving the semi-definite programming problem from equation (5) is in Appendix 6.2.5.

4.2 AMPL

AMPL is a modeling language for mathematical programming. Hence, there will be three different types of files to solve a mathematical program: model file, data file, and run file. Model file is where the user writes their model for the mathematical program. Data file is where the user have their data for the mathematical program. Run file is the file which the user will run to get solution for the mathematical programs.

Calling GUROBI, CPLEX, and XPRESS in AMPL

First, it is necessary to create a model, data, and run files. In the model file, the necessary objective function with the constraints should be included. In the data file, the necessary data for the problem should be included. In the run file, the specific solver will be called to solve the problem. To solve the Linear Programming problem from equation (1), we call the CPLEX solver. The following command in the run file should be included to solve using CPLEX.

```
option solver cplex;
```

Now to use other solvers such as XPRESS and GUROBI, one will have to call the specific solvers by changing the code above. Below is the command for calling the solvers XPRESS and GUROBI respectively.

```
option solver xpress;

option solver gurobi;
```

The necessary model and run files are in Appendix 6.6.1 for the linear programming problem from equation (1) with the solution which matches up with what we acquired from the Python section. Since there was no data for this problem, there was no data file for it. However, if one has data for the program, then they will have to include data file for the program. Similarly, to solve integer programming problem from equation (2), the following command should be altered in the model file from Linear Programming problem code in Appendix 6.6.1.

```
var x1 >= 0 integer;
var x2 >= 0 integer;
```

Basically, the continuous variables are now restricted to integers. The associated model and run files with solvers CPLEX, GUROBI, and XPRESS are in Appendix 6.6.2.

Now, to solve quadratic programming problem from equation (3), the associated model and run files are created using the objective function and the constraints from the equation (3). The associated model and run files with solvers CPLEX, GUROBI, and XPRESS are in Appendix 6.6.3.

Finally, to solve the nonlinear programming problem from equation (4), the associated model and run files are created using the nonlinear objective function and the constraints from the equation (4). For the solver, it is essential to use Minos because CPLEX, GUROBI, and XPRESS do not support nonlinear programming problems. The associated model and rule files with the solver Minos are in Appendix 6.6.4.

4.3 MATLAB

Since IBM CPLEX is installed using the steps in section 2.1, now to let MATLAB know that CPLEX solver is on the computer, one has to enter the following commands in MATLAB IDE.

```
>> addpath('<CPLEX install dir>\cplex\matlab\x64_win64')
>> savepath
```

Now, MATLAB knows that the system has CPLEX, so CPLEX can be used to solve mathematical programs. Similarly, to let MATLAB know that GUROBI solver is on the computer, the following necessary commands should be entered in MATLAB IDE.

```
>> cd c:/gurobi811/win64/matlab
>> gurobi_setup
```

Now, MATLAB knows that the system has GUROBI; it can be used to solve different types of mathematical programs. Likewise, to let MATLAB know that XPRESS solver is in the system, the following commands should be entered in MATLAB IDE.

```
>> addpath ('c:\xpressmp\matlab')
>> savepath
```

Now, MATLAB knows that XPRESS solver is in the system, so it can be used to solve different types of mathematical programs. All the solvers are set to solve mathematical programs, so these solvers will be called to solve specific mathematical programs.

Calling CPLEX, GUROBI, and XPRESS in MATLAB

Now, let us solve a linear programming problem, equation (1) from Appendix 6.1 using MATLAB. First of all, let us use CPLEX solver to solve the linear program. To be more specific, to solve linear programming problem, "cplexlp" function should be used. Thus, the structure of the solver function would look like

```
x=cplexlp(problem)
```

where the problem would be structured with the objective coefficients, constraints coefficients, and the bounds of the decision variables. Doing that with equation (1) from Appendix 6.1, one would obtain the program which is in Appendix 6.4.1 and the corresponding result which matches with the results in Python and AMPL sections.

The model setup for gurobi is slightly different than CPLEX. The model is set up first, and then gurobi is used to solve it. One would set up the model in a similar way like CPLEX, but then to solve using GUROBI, the following command is used.

```
result = gurobi(model);
```

However, there is a full program code in Appendix 6.4.1 for linear programming problem from equation (1) in Appendix 6.1 with the result which matches up with the result of CPLEX. Similarly, as in CPLEX, one can use the same problem structure to solve the LP using XPRESS. However, in this case, one uses "xprslp" function as the solver, as shown in the following command.

```
x=xprslp(problem);
```

Following is the table of XPRESS solvers for specific mathematical programs:

Table 1: XPRESS Solver Commands for Specific Mathematical Programs

Xpress MATLAB Function	**Mathematical Problem**
xprslp	Solve linear programming problems
xprsbip	Solve binary integer programming problems
xprsmip	Solve mixed integer linear programming problems
xprsqp	Solve quadratic programming problems

Now, for the integer programming problems from equation (2) in Appendix 6.1, everything from LP program code stays the same except the variable type. In this case, variables are restricted to integers. For CPLEX, one adds in a line of code below,

```
ctype='II'
```

which means that both decision variables are now integers. For GUROBI, we add in a line of code below,

```
model.vtype='I'
```

which means that all the decision variables are now integers. For XPRESS, one should add in a line of code that was added in CPLEX above. Now, this would output the integer solutions to our program. Each of the solvers gives the same result. The full mathematical program with code is in Appendix 6.4.1. Now, it is time to solve the quadratic programming problem from equation (3) in Appendix 6.1 using CPLEX, GUROBI, and XPRESS. Setting up a quadratic problem can be tricky. CPLEX and XPRESS can take quadratic programs with the objective of the form $\frac{1}{2}x^TQx + fx$. According to our problem from equation (3), our

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, Q = \begin{bmatrix} 6 & 2 \\ 2 & 2 \end{bmatrix}, f = \begin{pmatrix} 1 & 6 \end{pmatrix}$$

It is not so obvious to get Q. In order to get Q, little bit of matrix algebra is required. The following has to be true, so we will need to find the Q that would make this following statement true

$$\frac{1}{2}\begin{pmatrix} x_1 & x_2 \end{pmatrix} Q \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + f \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 3x_1^2 + 2x_1x_2 + x_2^2 + x_1 + 6x_2$$

Observing, $f = \begin{pmatrix} 1 & 6 \end{pmatrix}$

So, we need to find Q such that

$$\frac{1}{2}\begin{pmatrix} x_1 & x_2 \end{pmatrix} Q \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 3x_1^2 + 2x_1x_2 + x_2^2$$

That is only possible if $Q = \begin{bmatrix} 6 & 2 \\ 2 & 2 \end{bmatrix}$

Now, the main ingredients of the quadratic programming problem are found, so the next steps would be to combine the constraints and execute the program. The sample code for the quadratic problem from equation (3) in Appendix 6.1 is in Appendix 6.4.1 with the result which matches up with what we obtained in Python and AMPL sections.

There are also plenty of built-in optimization solvers in MATLAB which can solve linear, integer, and quadratic programming problems. The built-in solvers are linprog, intlinprog, and quadprog. There are sample codes for solving equation (1),(2), and (3) in Appendix 6.4.1. We still get the same result using these built-in solvers.

There is no convenient way of approaching nonlinear and semi-definite programming problems in MATLAB, so let us leave nonlinear and semi-definite programming problems in MATLAB.

4.4 C++

We have installed all the solvers from section 2.1. Now, it is essential to let C++ know that these solvers exist on the computer. For that, we will have to include specific solver's .h file in the program.

Calling CPLEX, GUROBI, and XPRESS in C++

First of all, to let C++ in Visual Studio know that the computer has CPLEX solver, one will have to follow the procedure given in this link:

`https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.3/ilog.odms.cplex.help/CPLEX/GettingStarted/topics/set_up/Windows.html`

Now once the C++ knows that the computer has CPLEX, we can then use CPLEX to solve our mathematical programs. Before we write the model, we should have the command

```
#include <ilcplex/ilocplex.h>
```

in the .cpp file. Whenever we have to use C++ classes in the CPLEX Concert Library, we begin with "Ilo", for example, "IloNumVar" for a numerical variable. Now, to build the model we start with "IloEnv env;" which creates the CPLEX environment which takes care of all the necessary stuffs including memory management for CPLEX objects. Then, we build the model using "IloModel model(env);" and include all the decision variables and constraints in it. Then, we solve it using "cplex.solve()" function. There is a sample code for solving linear program from equation (1) in Appendix 6.3.1. We use exactly the same function to build the model for integer programming problem from equation (2) and quadratic programming problem from equation (3). The sample codes for those linear, integer, and quadratic programming problems are in Appendix 6.3.1, 6.3.2, and 6.3.3 respectively.

Similarly, we will let C++ in Visual Studio know that our computer has solver Gurobi. To do that, we will have to follow the procedure here in this link: `https://support.gurobi.com/hc/en-us/articles/360013194392-How-do-I-configure-a-ne`

Now once the C++ knows that the computer has Gurobi, we can then use Gurobi to solve our mathematical programs. Before we begin building the model, we should have the command

```
#include "gurobi_c++.h"
```

in .cpp file. Now to build the model, we start with "GRBEnv env = GRBEnv():" which creates the GUROBI environment. Then, we create the model using "GRBModel(env);" and variables using "GRBVar;". Then we include the constraints and solve the problem using the solver by the command "model.optimize();". This will solve the mathematical program that we proposed in the model. There is a sample code for solving the linear program from equation (1) in Appendix 6.3.1. It is a very similar procedure for equation (2) and (3). Hence, the sample code for linear, integer, and quadratic programming problems are in Appendix 6.3.1, 6.3.2, and 6.3.3, respectively. There is no proper instruction on how to set up the C++ interface for XPRESS by the FICO XPRESS, so we skip XPRESS part for C++.

4.5 R

R is a language and environment for statistical computing and graphics (Rproject.org). It provides with wide varieties of statistical and graphical techniques such as linear modeling, nonlinear modeling, time series analysis, and other statistical analysis. It is a free software under the terms of General Public License (GNU) in source code form. However, sometimes we may have to optimization problems in R. So for our purpose, we will be just looking for built-in optimization solvers in R. We will learn how to install and use them to solve mathematical programs.

Built-in Optimization Solvers

Package 'lpSolve'

It is a software package in R for solving linear, integer, and mixed integer programs. More information about the package is found in following link:

`https://cran.r-project.org/web/packages/lpSolve/lpSolve.pdf`

We use the following command in R to install the lpSolve package:

```
install.packages("lpSolve")
```

Now, we use lpSolve to solve a linear program from equation (1). The sample code for that specific program is found in Appendix 6.5.1. We use the same solver package to solve integer programming problem from equation (2) in Appendix 6.1. The only difference for integer program is that we restrict the variables to integer, so we add "all.int=TRUE" in the lp() function. The associated sample code for integer program from equation (2) is in Appendix 6.5.2.

Package 'quadprog' To solve the quadratic programming problem from equation (3) in Appendix 6.1, we need to install the 'quadprog' package in R. To install that in R, we run this following command in R cmd.

```
install.packages("quadprog")
```

Like we said before, quadratic programming problem is tricky to enter in the solver. First we have to put our quadratic programming problem in the form $-d^T b + \frac{1}{2} b^T D b$ with constraints $A^T b \geq b_0$ in R. In our case,

$$D = \begin{bmatrix} 6 & 2 \\ 2 & 2 \end{bmatrix}, b_0 = 4, A = \begin{bmatrix} 2 & 3 \end{bmatrix}, d = \begin{bmatrix} -1 \\ -6 \end{bmatrix}$$

Now, we implement these in the model to solve the problem using quadprog solver. The associated sample code for this problem is in Appendix 6.5.3.

4.6 XPRESS-MOSEL

XPRESS-MOSEL is a platform where we can solve the optimization problem. When we get the XPRESS solver from FICO XPRESS as an academia license, XPRESS IVE comes with it, where we build programming models and solve it. When one opens the XPRESS IVE platform, they get to create the model in an XPRESS-MOSEL file. It comes with a sample model every time one creates a new file, which makes it easy to navigate through the program and make a model. First of all, we write the name of the model and the solver we need to solve the problem, to solve equation (1) from Appendix 6.1, we use the following command

```
model "Linear Program Example" % create name of model
uses "mmxprs"; % using xpress solver
```

Then, we declare our variables, constraints, and objective function. In the end, we display the solution. The associated sample code is in Appendix 6.7.1. Everything stays same for integer programming problem from equation (2) in Appendix 6.1, but we restrict the variables to integer by the following command.

```
x1 is_integer; x2 is_integer
```

The full sample code is in Appendix 6.7.2. For the quadratic programming problem from equation (3) in Appendix 6.1, we add specific quadratic solver in the code, which we can see below.

```
uses "mmxprs", "mmquad"
```

Also, we declare the variables in an array, but everything else is as similar as we had in for Linear Programming problem. The associated sample code for quadratic programming problem from equation (3) is in Appendix 6.7.3.

XPRESS-MOSEL can also solve nonlinear programming problem. So, we solve equation (4) from Appendix 6.1 using xpress nonlinear solver. To access this, we have to include the following command.

```
uses "mmxnlp";
```

This allow us to solve the nonlinear programming problem from equation (4) in Appendix 6.1. The associated sample code for this nonlinear programming problem is in Appendix 6.7.4.

5 Solvers for specific mathematical programs

Mainly, the solvers that we studied were CPLEX, Gurobi, and Xpress. There are many other optimization solvers that can solve specific mathematical programs, so there is no such thing as the "best" solver. Different solvers are capable of solving different kinds of optimization problems. Because of the market competitions influenced by the makers of the commercial solvers, it is not easy to tell which one is the best.

5.1 Solvers for Linear Programs

Any of the solvers CPLEX, Gurobi, Xpress, CBC, CLP, MINOS, etc. can solve linear programming problems. There are many online optimization solvers' discussions which say that CPLEX can take up to millions of variables and constraints. Similarly, other solvers such as Gurobi, CBC, CLP, GLPK, Xpress, LPSolve, etc. can solve programs with several millions of variables as per the discussions. However, there are no specific numbers on the limit by the makers of these solvers. Hence almost all of the optimization solvers can solve linear programs.

5.2 Solvers for Integer programs

Gurobi,CPLEX,and Xpress are all good for Integer programming problems. There are several other optimization solvers out there which can solve integer programs. However, CPLEX cannot solve nonlinear integer programs, it can solve linear programs, mixed integer programs, quadratic programs and quadratically constrained programs. Similarly, gurobi can only solve linear programs, mixed integer programs, quadratic programs, mixed-integer quadratic programs, mixed-integer quadratically constrained programs.

5.3 Solvers for Quadratic Programs

Gurobi, CPLEX, and Xpress are all good for quadratic programs. On the other hand, CBC and CLP cannot solve quadratic problems. However, there are other optimization solvers that are not listed here which can solve quadratic programs.

5.4 Solvers for Nonlinear Programming

Gurobi, CPLEX, CBC, and CLP cannot solve nonlinear programs. XPRESS, IPOPT, and MINOS can solve nonlinear programming problems.

5.5 Solvers for Semidefinite

None of the solvers we studied here can solve semi-definite programs. Semidefinite programs can be solved by the solvers such as: LMILAB, MOSEK, PENBMI, PENSDP,sdp in CVXOPT, etc. Some are free for academic purposes but some requires you to purchase the solver. MOSEK and PENDSP are free for academia.

6 Appendix

6.1 Examples of types of Mathematical Program

Linear Programming Problem

$$\begin{aligned} \textbf{max } 30x_1 + 20x_2 \\ \textbf{subject to } x_1 + 2x_2 \leq 8 \\ 2x_1 + x_2 \leq 8 \\ x_1, x_2 \geq 0 \end{aligned} \tag{1}$$

Integer Programming Problem

$$\begin{aligned} \textbf{max } & 30x_1 + 20x_2 \\ \textbf{subject to } & x_1 + 2x_2 \leq 8 \\ & 2x_1 + x_2 \leq 8 \\ & x_1, x_2 \geq 0 \\ & x_1, x_2 \in \mathbb{Z}^+ \end{aligned} \tag{2}$$

Quadratic Programming Problem

$$\begin{aligned} \textbf{max } & 3x_1^2 + x_2^2 + 2x_1x_2 + x_1 + 6x_2 + 2 \\ \textbf{subject to } & 2x_1 + 3x_2 \geq 4 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{3}$$

Nonlinear Programming Problem

$$\begin{aligned} \textbf{max } & ln(x_1 + 1) + 2x_2 \\ \textbf{subject to } & 2x_1 + x_2 \leq 3 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{4}$$

Semidefinite Programming Problem[1]

min $x_1 - x_2 + x_3$

[1] This semidefinite problem is exactly from the CVXOPT user guide referenced in [4]

$$x_1 \begin{bmatrix} -7 & -11 \\ -11 & 3 \end{bmatrix} + x_2 \begin{bmatrix} 7 & -18 \\ -18 & 8 \end{bmatrix} + x_3 \begin{bmatrix} -2 & -8 \\ -8 & 1 \end{bmatrix} \preceq \begin{bmatrix} 33 & -9 \\ -9 & 26 \end{bmatrix} \tag{5}$$

$$x_1 \begin{bmatrix} -21 & -11 & 0 \\ -11 & 10 & 8 \\ 0 & 8 & 5 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 10 & 16 \\ 10 & -10 & -10 \\ 16 & -10 & 3 \end{bmatrix} + x_3 \begin{bmatrix} -5 & 2 & -17 \\ 2 & -6 & 8 \\ -17 & 8 & 6 \end{bmatrix} \preceq \begin{bmatrix} 14 & 9 & 40 \\ 9 & 91 & 10 \\ 40 & 10 & 15 \end{bmatrix}$$

6.2 Example Programs with Python

6.2.1 Linear Programming

Using PuLP library (CPLEX): [2]

```
from sympy import * # import sympy library
from pulp import *  # import pulp library
prob = LpProblem ("Simple", LpMaximize) # setup the model
x1 = LpVariable ("x1", lowBound=0) # initialize the variables
x2 = LpVariable ("x2", lowBound=0)
prob += 30*x1+20*x2  # setup objective function
prob += 2*x1+x2 <= 8 # add your constraints
prob += x1 + 2*x2 <= 8
status = prob.solve(CPLEX()) # Use CPLEX solver to solve
value(x1), value (x2), value(prob.objective)
```

Result

```
Out[30]:
(2.6666666666666665, 2.666666666666667, 133.33333333333334)
```

Using Pyomo library (CPLEX) [3]

```
from pyomo.environ import * # module pyomo.environ is necessary
m = ConcreteModel() #using ConcreteModel() function to create the model
m.x1 = Var(bounds=(0,None))
```

[2]Sample code is used from PuLP 1.6.0 documentation referenced in [9]
[3]Sample codes is used from Pyomo website referenced in [16]

```
m.x2 = Var(bounds=(0,None))
m.o = Objective(expr=-(30*m.x1+20*m.x2)) # Setting Objective Function
m.c1 = Constraint(expr=m.x1+2*m.x2 <= 8) # Setting Constraints
m.c2 = Constraint(expr=2*m.x1+m.x2 <= 8)
solver = SolverFactory('cplex') # Calling the solver
status = solver.solve(m) # Solve
print("Status = %s" % status.solver.termination_condition)
print("%s = %f" % (m.x1, value(m.x1)))
print("%s = %f" % (m.x2, value(m.x2)))
print("Objective = %f" % value(m.o))
```

Result

```
Status = optimal
x1 = 2.666667
x2 = 2.666667
Objective = -133.333333
```

6.2.2 Integer Programming

Using PuLP library (CPLEX)

```
from sympy import *
from pulp import *
prob = LpProblem ("Simple", LpMaximize)        # Integer Linear Programming
x1 = LpVariable ("x1", lowBound=0, cat='Integer') # Domain Restricted to Integers
x2 = LpVariable ("x2", lowBound=0, cat='Integer')
prob += 30*x1+20*x2
prob += 2*x1+x2 <= 8
prob += x1 + 2*x2 <= 8
status = prob.solve(CPLEX())
value(x1), value (x2), value(prob.objective)
```

Result

```
(3.0, 2.0, 130.0)
```

Using Pyomo Library (CPLEX)

```
import numpy as np
from pyomo.environ import *
m = ConcreteModel()
m.x1 = Var(bounds=(0,None),within=NonNegativeIntegers)
```

```
m.x2 = Var(bounds=(0,None),within=NonNegativeIntegers)
m.o = Objective(expr=-(30*m.x1+20*m.x2))
m.c1 = Constraint(expr=m.x1+2*m.x2 <= 8)
m.c2 = Constraint(expr=2*m.x1+m.x2 <= 8)
solver = SolverFactory('cplex')
status = solver.solve(m)
print("Status = %s" % status.solver.termination_condition)
print("%s = %f" % (m.x1, value(m.x1)))
print("%s = %f" % (m.x2, value(m.x2)))
print("Objective = %f" % value(m.o))
```

Result

```
Status = optimal
x1 = 3.000000
x2 = 2.000000
Objective = -130.000000
```

6.2.3 Quadratic Programming

Using Pyomo library (CPLEX)

```
import numpy as np
from pyomo.environ import *
m = ConcreteModel()
m.x1 = Var(bounds=(0,None))
m.x2 = Var(bounds=(0,None))
m.o = Objective(expr=3*m.x1*m.x1+m.x2*m.x2+2*m.x1*m.x2+m.x1+6*m.x2+2)
m.c = Constraint(expr=2*m.x1+3*m.x2 >= 4)
solver = SolverFactory('cplex')
status = solver.solve(m)
print("Status = %s" % status.solver.termination_condition)
print("%s = %f" % (m.x1, value(m.x1)))
print("%s = %f" % (m.x2, value(m.x2)))
print("Objective = %f" % value(m.o))
```

Result

```
Status = optimal
x1 = 0.500000
x2 = 1.000000
Objective = 11.250000
```

6.2.4 Nonlinear Programming

Using Pyomo Library IPOPT Solver [4]

```
import numpy as np
from pyomo.environ import *
m = ConcreteModel()
m.x1 = Var(bounds=(0,None))
m.x2 = Var(bounds=(0,None))
m.o = Objective(expr=-(log(m.x1+1)+2*m.x2))
m.c = Constraint(expr=2*m.x1+m.x2 <= 3)
solver = SolverFactory('IPOPT')
status = solver.solve(m)
print("Status = %s" % status.solver.termination_condition)
print("%s = %f" % (m.x1, value(m.x1)))
print("%s = %f" % (m.x2, value(m.x2)))
print("Objective = %f" % value(m.o))
```

Result

```
Status = optimal
x1 = 0.000000
x2 = 3.000000
Objective = -6.000000
```

6.2.5 Semidefinite Programming

Using cvxopt library with SDP Solver[5]

```
from cvxopt import matrix, solvers
c = matrix([1.,-1.,1.])
G = [ matrix([[-7., -11., -11., 3.],
              [ 7., -18., -18., 8.],
              [-2., -8., -8., 1.]])]
G += [ matrix([[-21., -11., 0., -11., 10., 8., 0., 8., 5.],
               [ 0., 10., 16., 10., -10., -10., 16., -10., 3.],
               [ -5., 2., -17., 2., -6., 8., -17., 8., 6.]]) ]
h = [ matrix([[33., -9.], [-9., 26.]]) ]
h += [ matrix([[14., 9., 40.], [9., 91., 10.], [40., 10., 15.]]) ]
sol = solvers.sdp(c, Gs=G, hs=h)
```

[4]Sample code used from referenced site in [15]

[5]This sample code is exactly from the CVXOPT user guide referenced in [4]

```
print(sol['x'])
print(sol['zs'][0])
print(sol['zs'][1])
```

Result

```
Optimal solution found.
[-3.68e-01]
[ 1.90e+00]
[-8.88e-01]

[ 3.96e-03 -4.34e-03]
[-4.34e-03  4.75e-03]

[ 5.58e-02 -2.41e-03  2.42e-02]
[-2.41e-03  1.04e-04 -1.05e-03]
[ 2.42e-02 -1.05e-03  1.05e-02]
```

6.3 Example Programs with C++

6.3.1 Linear Programming

Using CPLEX [6]

```
#include <ilcplex/ilocplex.h>
int
main()
{
IloEnv env;
IloModel model(env);
IloNumVar x1 = IloNumVar(env, 0, IloInfinity);
IloNumVar x2 = IloNumVar(env, 0, IloInfinity);
model.add(IloMaximize(env, 30*x1 + 20*x2));
IloRangeArray constraints(env);
constraints.add(2 * x1 + x2 <= 8);
constraints.add(x1 + 2 * x2 <= 8);
model.add(constraints);
IloCplex cplex(model);
cplex.solve();
```

[6]Sample codes of C++ API for CPLEX are used from the example package that comes with the main IBM CPLEX solver package for both linear,integer, and quadratic programming

```
using std::cout; using std::endl;
cout << "Solutions" << endl;
cout << cplex.getValue(x1) << endl;
cout << cplex.getValue(x2) << endl;
cout << cplex.getObjValue() << endl;
}
```

(Result)

```
Iteration:     1   Dual infeasibility =            0.000000
Iteration:     2   Dual objective     =          133.333333
Solutions
2.66667
2.66667
133.333
```

Using GUROBI [7]

```
#include "gurobi_c++.h"
int main(int argc, char *argv[])
{
try
{
GRBEnv env = GRBEnv();
GRBModel model = GRBModel(env);
// Set Variables
GRBVar x1 = model.addVar(0.0, GRB_INFINITY,0.0, GRB_CONTINUOUS, "x1");
GRBVar x2 = model.addVar(0.0, GRB_INFINITY, 0.0, GRB_CONTINUOUS, "x2");
model.update();
// Set Objective
model.setObjective(30*x1 + 20*x2, GRB_MAXIMIZE);
// Set constraints:
model.addConstr(2*x1 + x2 <= 8, "c0");
model.addConstr(x1 + 2*x2 <= 8, "c1");
// Run the optimizion
model.optimize();
std::cout << x1.get(GRB_StringAttr_VarName) << " "
<< x1.get(GRB_DoubleAttr_X) << std::endl;
std::cout << x2.get(GRB_StringAttr_VarName) << " "
```

[7]Sample codes of C++ API for GUROBI are used from the example package that comes with the main GUROBI solver package for both linear,integer,and quadratic programming

```
<< x2.get(GRB_DoubleAttr_X) << std::endl;
std::cout << "Obj: " << model.get(GRB_DoubleAttr_ObjVal) << std::endl;
}
catch (GRBException e)
{
std::cout << "Error code = "
<< e.getErrorCode()
<< std::endl;
std::cout << e.getMessage() << std::endl;
}
catch (...)
{
std::cout << "Exception during optimization"
<< std::endl;
}
return 0;
}
```

Result

```
Solved in 2 iterations and 0.00 seconds
Optimal objective  1.333333333e+02
x1 2.66667
x2 2.66667
Obj: 133.333
```

6.3.2 Integer Programming

Using CPLEX

```
#include <ilcplex/ilocplex.h>
int main()
{
IloEnv env;
IloModel model(env);
IloNumVar x1 = IloNumVar(env, 0, IloInfinity, IloNumVar::Int);
IloNumVar x2 = IloNumVar(env, 0, IloInfinity, IloNumVar::Int);
model.add(IloMaximize(env, 30*x1 + 20*x2));
IloRangeArray constraints(env);
constraints.add(2 * x1 + x2 <= 8);
constraints.add(x1 + 2 * x2 <= 8);
model.add(constraints);
IloCplex cplex(model);
cplex.solve();
using std::cout; using std::endl;
cout << "Solutions" << endl;
cout << cplex.getValue(x1) << endl;
cout << cplex.getValue(x2) << endl;
cout << cplex.getObjValue() << endl;
}
```

Result

```
Total (root+branch&cut) =    0.03 sec. (0.02 ticks)
Solutions
3
2
130
```

Using GUROBI

```
#include "gurobi_c++.h"
int main(int argc, char *argv[])
{
try
{
GRBEnv env = GRBEnv();
GRBModel model = GRBModel(env);
// Set Variables // This is the Integer restriction for domain
GRBVar x1 = model.addVar(0.0, GRB_INFINITY,0.0, GRB_INTEGER, "x1");
GRBVar x2 = model.addVar(0.0, GRB_INFINITY, 0.0, GRB_INTEGER, "x2");
model.update();
// Set Objective
model.setObjective(30*x1 + 20*x2, GRB_MAXIMIZE);
// Set constraints:
model.addConstr(2*x1 + x2 <= 8, "c0");
model.addConstr(x1 + 2*x2 <= 8, "c1");
// Run the optimizion
model.optimize();
std::cout << x1.get(GRB_StringAttr_VarName) << " "
<< x1.get(GRB_DoubleAttr_X) << std::endl;
std::cout << x2.get(GRB_StringAttr_VarName) << " "
<< x2.get(GRB_DoubleAttr_X) << std::endl;
std::cout << "Obj: " << model.get(GRB_DoubleAttr_ObjVal) << std::endl;
}
catch (GRBException e)
{
std::cout << "Error code = "
<< e.getErrorCode()
<< std::endl;
std::cout << e.getMessage() << std::endl;
}
catch (...)
{
std::cout << "Exception during optimization"
<< std::endl;
}
return 0;
}
```

Result

```
Optimal solution found (tolerance 1.00e-04)
Best objective 1.300000000000e+02, best bound 1.300000000000e+02, gap 0.0000%
x1 3
x2 2
Obj: 130
```

6.3.3 Quadratic Programming

Using GUROBI

```
#include "gurobi_c++.h"
using namespace std;
int main(int   argc,
char *argv[])
{
try {
GRBEnv env = GRBEnv();
GRBModel model = GRBModel(env);
// Create variables
GRBVar x1 = model.addVar(0.0, 1.0, 0.0, GRB_CONTINUOUS, "x1");
GRBVar x2 = model.addVar(0.0, 1.0, 0.0, GRB_CONTINUOUS, "x2");
// Set objective
GRBQuadExpr obj = 3 * x1*x1 + x2*x2 + 2 * x1*x2 + x1 + 6 * x2 + 2;
model.setObjective(obj);
// set constriants
model.addConstr(2 * x1 + 3 * x2 >= 4, "c0");
// Optimize model
model.optimize();
cout << x1.get(GRB_StringAttr_VarName) << " "
<< x1.get(GRB_DoubleAttr_X) << endl;
cout << x2.get(GRB_StringAttr_VarName) << " "
<< x2.get(GRB_DoubleAttr_X) << endl;
cout << "Obj: " << model.get(GRB_DoubleAttr_ObjVal) << endl;
// Change variable types to integer
x1.set(GRB_CharAttr_VType, GRB_CONTINUOUS);
x2.set(GRB_CharAttr_VType, GRB_CONTINUOUS);
// Optimize model
model.optimize();
cout << x1.get(GRB_StringAttr_VarName) << " "
<< x1.get(GRB_DoubleAttr_X) << endl;
cout << x2.get(GRB_StringAttr_VarName) << " "
<< x2.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;
}
```

Result

```
Optimal objective 1.12500000e+01
x1 0.500042
x2 0.999972
Obj: 11.25
```

Using CPLEX

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN
static void
populatebyrow(IloModel model, IloNumVarArray var, IloRangeArray con);
int main(int argc, char **argv)
{
IloEnv   env;
try {
IloModel model(env);
IloNumVarArray var(env);
IloRangeArray con(env);
populatebyrow(model, var, con);
IloCplex cplex(model);
if (!cplex.solve()) {
env.error() << "Failed to optimize LP" << endl;
throw(-1);
}
IloNumArray vals(env);
env.out() << "Solution status = " << cplex.getStatus() << endl;
env.out() << "Solution value  = " << cplex.getObjValue() << endl;
cplex.getValues(vals, var);
env.out() << "Values        = " << vals << endl;
cplex.getSlacks(vals, con);
env.out() << "Slacks        = " << vals << endl;
cplex.getDuals(vals, con);
env.out() << "Duals         = " << vals << endl;
cplex.getReducedCosts(vals, var);
env.out() << "Reduced Costs = " << vals << endl;
cplex.exportModel("qpex1.lp");
}
catch (IloException& e) {
cerr << "Concert exception caught: " << e << endl;
}
catch (...) {
cerr << "Unknown exception caught" << endl;
}
env.end();
return 0;
}
static void
```

```
populatebyrow(IloModel model, IloNumVarArray x, IloRangeArray c)
{
IloEnv env = model.getEnv();
x.add(IloNumVar(env, 0.0, IloInfinity));
x.add(IloNumVar(env));
model.add(IloMinimize(env, 3 * x[0]*x[0] + x[1]*x[1] + 2 *x[0]*x[1] + x[0] + 6 * x[1]
));
c.add(-2 * x[0] - 3 * x[1] <= -4);
model.add(c);
}
```

Result

```
Solution status = Optimal
Solution value  = 11.25
Values          = [0.5, 1]
```

6.4 Example Programs with MATLAB

6.4.1 Linear Programming

Using CPLEX [8]

```
f = [30, 20];
Aeq = [ 1, 2;
      2, 1];
beq = [8 8];
lb = zeros(2,1);
ub = [Inf;Inf];
[x,fval] = cplexlp(f,[],[],Aeq,beq,lb,ub)
```

Result

```
>> cplexlpexample
x = 2.6667
    2.6667
fval =
  133.3333
```

Using GUROBI [9]

[8]Sample code from IBM CPLEX solver package under MATLAB example package is used for linear, integer, and quadratic programming

[9]Sample code from GUROBI solver package under MATLAB example package is used for linear, integer, and quadratic programming

```
model.A =sparse([1 2;2 1]);
model.obj =[30 20];
model.modelsense = 'Max';
model.rhs = [8 8];
model.sense = ['<' '<'];
result = gurobi(model);
disp(result.objval);
disp(result.x);
```

Result

```
>> gurobimatlabex
Optimal objective  1.333333333e+02
  133.3333
    2.6667
    2.6667
```

Using XPRESS [10]

```
f = [30; 20];
A = [ 1 2
      2 1];
b = [8,8];
lb = zeros(2,1);
x=xprslp(-f,A,b); % xprslp is for linear programs
disp(x)
```

Result

```
>> xpressexample
Final objective                       : -1.333333333333333e+02
  Max primal violation      (abs/rel) :       0.0 /       0.0
  Max dual violation        (abs/rel) :       0.0 /       0.0
  Max complementarity viol. (abs/rel) :       0.0 /       0.0
All values within tolerances
    2.6667
    2.6667
```

[10]Sample codes from XPRESS solver package under MATLAB example package is used for linear, integer, and quadratic programming

Using Built-in linprog

```
f = [30, 20];
Aeq = [ 1, 2;
      2, 1];
beq = [8 8];
lb = zeros(2,1);
ub = [Inf;Inf];
[x,fval] = linprog(f,[],[],Aeq,beq,lb,ub)
```

Result

```
>> linearprogrammatlabbuiltin
x = 2.6667
    2.6667
fval =
  133.3333
```

6.4.2 Integer Programming

Using CPLEX

```
f     = [-30 -20]';
   Aineq = [2 1;
      1 2];
   bineq = [8 8]';
   lb    = [0;    0];
   ub    = [inf; inf];
   ctype = 'II';
options = cplexoptimset;
options.Display = 'on';
x=cplexmilp (f, Aineq, bineq, [ ], [ ],...
      [ ], [ ], [ ], lb, ub, ctype, [ ], options);
disp(x)
```

Result

```
>> cplexintegerprogram
Total (root+branch&cut) =    0.00 sec. (0.02 ticks)
     3
     2
```

Using GUROBI

```
model.A =sparse([1 2;2 1]);
model.obj =[30 20];
model.modelsense = 'Max';
model.rhs = [8 8];
model.sense = ['<' '<'];
model.vtype= 'I';         % Add variables to be integers
result = gurobi(model);
disp(result.objval);
disp(result.x);
```

Result

```
>> gurobiintegerprogram
Optimal solution found (tolerance 1.00e-04)
Best objective 1.300000000000e+02, best bound 1.300000000000e+02, gap 0.0000%
   130
     3
     2
```

Using XPRESS

```
f = [30; 20];
A = [ 1 2
      2 1];
b = [8,8];
lb = zeros(2,1);
ctype='II';
x=xprsmip(-f,A,b,[],ctype); % xprsmip is for linear programs
disp(x)
```

Result

```
>> integerprogram
Best integer solution found is  -130.000000
Best bound is  -130.000000
Uncrunching matrix
     3
     2
```

Using Built-in intlinprog

```
f = [30; 20];
intcon = [1,2];
```

```
A = [1, 2; 2, 1];
b = [8;8];
lb = zeros(2,1);
ub = [Inf;Inf];
[x,fval] = intlinprog(-f,intcon,A,b,[],[],lb,ub)
```

Result

```
x =3.0000
   2.0000
fval =
  -130
```

6.4.3 Quadratic Programming

Using GUROBI

```
names = {'x1', 'x2'};
model.varnames = names;
model.Q = sparse([3 1 ; 1 1]);
model.A = sparse([2 3]);
model.obj = [1 6];
model.rhs = [4];
model.sense = '>';
gurobi_write(model, 'qp.lp');
results = gurobi(model);
for v=1:length(names)
    fprintf('%s %f\n', names{v}, results.x(v));
end
fprintf('Obj = %f \n', results.objval+2);
```

Result

```
>> gurobiquadprogram
x1 0.500000
x2 1.000000
Obj = 11.250000
```

Using CPLEX

```
H  = [6 2; 2 2];
f  = [ 1 6]';
Aineq  = [-2 -3];
```

```
bineq  = -4;
lb = [ 0   0 ];
ub = [Inf; Inf];
[x, fval, exitflag, output] = cplexqp (H, f, Aineq, bineq, [ ], [ ], lb,...
      ub, [ ]);
 fprintf ('Obj = %f \n', fval+2);
 disp ('x1 x2 =');
 disp (x');
```

Result

```
>> exampleqpcplex
Obj = 11.250000
x1 x2 =
    0.5000    1.0000
```

Using XPRESS

```
 H  = [6 2; 2 2];
   f  = [ 1; 6];
   Aineq  = [2 3];
   bineq  = [4];
   rtype='G';
   lb = zeros(2,1);
   ub = [Inf;Inf];
   [x,fval]= xprsqp (H, f, Aineq, bineq, rtype, lb,...
      []);
   disp('x1 x2 =');
   disp (x);
   fprintf('Obj: %f\n',fval+2);
```

Result

```
x1 x2 =
    0.5000
    1.0000
Obj: 11.250001
```

Using Built-in quadprog

```
H = sparse([6 2;
    2 2]);
f = sparse([1 6]);
```

```
Aeq = sparse([-2 -3]);
beq = [-4];
lb = zeros(2,1);
ub = [Inf;Inf];
[x,fval]= quadprog(H,f,[],[],Aeq,beq,lb,ub);
disp(x);
fval=fval+2
```

Result

```
0.5000
1.0000
fval =11.2500
```

6.5 Example Programs with R

6.5.1 Linear Programming

Using lpSolve [11]

```
> require(lpSolve) # call the package
> ## Set the coefficients of the decision variables -> C
> C <- c(30, 20)
> # Create constraint martix B
> A <- matrix(c(1, 2,
+              2, 1
+              ), nrow=2, byrow=TRUE)
> # Right hand side for the constraints
> B <- c(8, 8)
> # Direction of the constraints
> constranints_direction  <- c("<=", "<=")
> # Find the optimal solution
> optimum <-  lp(direction="max",
+                objective.in = C,
+                const.mat = A,
+                const.dir = constranints_direction,
+                const.rhs = B)
```

Result

[11]Sample codes are used from the referenced website in [14] for linear and integer programming

```
> print(optimum$status)
[1] 0
> # Display the optimum values for x_1 and x_2
> best_sol <- optimum$solution
> names(best_sol) <- c("x_1", "x_2")
> print(best_sol)
     x_1      x_2
2.666667 2.666667
> # Check the value of objective function at optimal point
> print(paste("Total cost: ", optimum$objval, sep=""))
[1] "Total cost: 133.333333333333"
```

6.5.2 Integer Programming

Using lpSolve

```
require(lpSolve) # call the package
## Set the coefficients of the decision variables -> C
C <- c(30, 20)
# Create constraint martix B
A <- matrix(c(1, 2,
              2, 1
              ), nrow=2, byrow=TRUE)
# Right hand side for the constraints
B <- c(8, 8)
types <- c("I", "I")
# Direction of the constraints
constranints_direction  <- c("<=", "<=")
# Find the optimal solution
optimum <-  lp(direction="max",
               objective.in = C,
               const.mat = A,
               const.dir = constranints_direction,
              const.rhs = B,all.int=TRUE) ## all variables are integers
best_sol <- optimum$solution
names(best_sol) <- c("x_1", "x_2")
print(best_sol)
```

Result

```
x_1 x_2
  3   2
```

6.5.3 Quadratic Programming

Using quadprog [12]

```
require(quadprog)
Dmat       <- matrix(c(6,2,2,2),2,2)
dvec       <- c(-1,-6)
Amat       <- matrix(c(2,3))
bvec       <- c(4)
solve.QP(Dmat,dvec,Amat,bvec=bvec)$solution
(solve.QP(Dmat,dvec,Amat,bvec=bvec)$value)+2
```

Result

```
> solve.QP(Dmat,dvec,Amat,bvec=bvec)$solution
[1] 0.5 1.0
> (solve.QP(Dmat,dvec,Amat,bvec=bvec)$value)+2
[1] 11.25
```

6.6 Example Programs with AMPL

6.6.1 Linear Programming

Model file:

```
#decision variables
var x1 >= 0;
var x2 >= 0;
# objective function
maximize z : 30*x1+20*x2;
# constraints
s.t. m1: 2*x1+x2 <= 8;
s.t. m2: x1+2*x2 <= 8;
```

Run file for CPLEX:

```
reset;
model simplelp.mod;
option solver cplexamp;
solve;
display x1, x2;
```

[12]Sample code from reference [1] is used for quadratic programming

Result

```
ampl: reset;
ampl: include simplelp.run;
CPLEX 12.9.0.0: optimal solution; objective 133.3333333
2 dual simplex iterations (1 in phase I)
x1 = 2.66667
x2 = 2.66667
```

Run file for GUROBI:

```
reset;
model simplelp.mod;
option solver gurobi;
solve;
display x1, x2;
```

Result

```
ampl: reset;
ampl: include simplelp.run;
Gurobi 8.1.0: optimal solution; objective 133.3333333
2 simplex iterations
x1 = 2.66667
x2 = 2.66667
```

Run file for XPRESS:

```
reset;
model simplelp.mod;
option solver xpress;
solve;
display x1, x2;
```

Result

```
ampl: reset;
ampl: include simplelp.run;
XPRESS 8.5.10(33.01.12): Optimal solution found
Objective 133.3333333
2 simplex iterations
x1 = 2.66667
x2 = 2.66667
```

6.6.2 Integer Programming

Run file for CPLEX:

```
reset;
model simpleilp.mod;
option solver cplexamp;
solve;
display x1, x2;
```

Result

```
ampl: reset;
ampl: include simpleilp.run;
CPLEX 12.9.0.0: optimal integer solution; objective 130
2 MIP simplex iterations
0 branch-and-bound nodes
x1 = 3
x2 = 2
```

Run file for GUROBI:

```
reset;
model simpleilp.mod;
option solver gurobi;
solve;
display x1, x2;
```

Result

```
ampl: reset;
ampl: include simpleilp.run;
Gurobi 8.1.0: optimal solution; objective 130
3 simplex iterations
x1 = 3
x2 = 2
```

Run file for XPRESS:

```
reset;
model simpleilp.mod;
option solver xpress;
solve;
display x1, x2;
```

Result

```
ampl: reset;
ampl: include simpleilp.run;
XPRESS 8.5.10(33.01.12): Global search complete
Best integer solution found 130
4 integer solutions have been found
1 branch and bound node
No basis.
x1 = 3
x2 = 2
```

6.6.3 Quadratic Programming

Model file:

```
var x1 >= 0;
var x2 >= 0;

# objective function
minimize z : 3*(x1)^2+(x2)^2+2*x1*x2+x1+6*x2+2;
s.t. m1: 2*x1+3*x2 >= 4;
```

Run file for CPLEX:

```
reset;
model quadratic.mod;
option solver cplexamp;
solve;
display x1, x2;
```

Result

```
ampl: reset;
ampl: include quadratic.run;
CPLEX 12.9.0.0: optimal solution; objective 11.25000002
16 QP barrier iterations
No basis.
x1 = 0.5
x2 = 1
```

Run file for GUROBI:

```
reset;
model quadratic.mod;
option solver gurobi;
solve;
display x1, x2;
```

Result

```
ampl: reset;
ampl: include quadratic.run;
Gurobi 8.1.0: optimal solution; objective 11.25
10 barrier iterations
No basis.
x1 = 0.5
x2 = 1
```

Run file for XPRESS:

```
reset;
model quadratic.mod;
option solver xpress;
solve;
display x1, x2;
```

Result

```
ampl: reset;
ampl: include quadratic.run;
XPRESS 8.5.10(33.01.12): Optimal solution found
Objective 11.25000107
5 barrier iterations
No basis.
x1 = 0.500015
x2 = 0.99999
```

6.6.4 Nonlinear Programming

Using Minos Model File:

```
var x1 >=0;
var x2 >=0;
```

```
maximize z : log(x1+1)+2*x2;
s.t. con1: 2*x1+x2 <=3;
```

Run file:

```
reset;
model nonlinearprogram.mod;
option solver minos;
solve;
display x1, x2;
```

Result

```
ampl: reset;
ampl: include nonlinearprogram.run;
MINOS 5.51: optimal solution found.
1 iterations, objective 6
Nonlin evals: obj = 5, grad = 4.
x1 = 0
x2 = 3
```

6.7 Example Programs with XPRESS-MOSEL

6.7.1 Linear Programming

Using XPRESS-MOSEL IVE [13]

```
model "Linear Program Example"
uses "mmxprs";
declarations
x1, x2: mpvar
end-declarations
x1+2*x2 <=8
2*x1+x2<=8
maximise(30*x1+20*x2)
writeln("x1:",getsol(x1))
writeln("x1:",getsol(x2))
writeln("Optimal Objective Value: ",getobjval)
end-model
```

Result

[13]Sample codes from reference in [11] are used for linear, integer, quadratic, and non-linear programming

```
x1:2.666666667
x1:2.666666667
Optimal Objective Value: 133.3333333
```

6.7.2 Integer Programming

```
model "Integer Program Example"
uses "mmxprs";
declarations
x1, x2: mpvar
end-declarations
x1 is_integer; x2 is_integer
x1+2*x2 <=8
2*x1+x2<=8
maximise(30*x1+20*x2)
writeln("x1:",getsol(x1))
writeln("x1:",getsol(x2))
writeln("Optimal Objective Value: ",getobjval)
end-model
```

Result

```
x1:3
x1:2
Optimal Objective Value: 130
```

6.7.3 Quadratic Programming

```
model "Quadratic Program Example"
 uses "mmxprs", "mmquad"
 declarations
  x: array(1..2) of mpvar
  Obj: qexp
 end-declarations
 2*x(1) + 3*x(2) >= 4
 Obj:= 3*x(1)^2 + x(2)^2 + 2*x(1)*x(2) + x(1) + 6*x(2)+2
 minimize(Obj)
 writeln("Solution: ", getobjval)
 forall(i in 1..2) writeln(getsol(x(i)))
end-model
```

Result

```
Solution: 11.25000107
0.5000150247
0.9999901029
```

6.7.4 Nonlinear Programming

```
model "Nonlinear Program Example"
uses "mmxnlp";
declarations
x1,x2: mpvar
end-declarations
2*x1+x2<=3
maximise( log(x1+1)+2*x2)
writeln("Optimal solution: x1=",getsol(x1)," x2=",getsol(x2))
writeln("Optimal Objective: ", getobjval)
```

Result

```
Optimal solution: x1=0 x2=3
Optimal Objective: 6
```

References

[1] A Barcelona Berwin. Functions to solve quadratic programming problems. *Package quadprog*.

[2] FICO. Fico xpress community license. `https://content.fico.com/xpress-optimization-community-license`.

[3] Robert Fourer, David M Gay, and Brian W Kernighan. *AMPL: A mathematical programming language*. AT & T Bell Laboratories Murray Hill, NJ 07974, 1987.

[4] CVXOPT User Guide. Semidefinite programming. `https://cvxopt.org/userguide/coneprog.html#semidefinite-programming`.

[5] Susanne Heipcke. Xpress–mosel. In *Algebraic Modeling Systems*, pages 77–110. Springer, 2012.

[6] IBM.com. Ibm ilog cplex optimization studio free edition. `https://www.ibm.com/account/reg/us-en/signup?formid=urx-20028`.

[7] David G Luenberger, Yinyu Ye, et al. *Linear and nonlinear programming*, volume 2. Springer, 1984.

[8] Sonja Mars and Lars Schewe. An sdp-package for scip. Technical report, Technical Report 08/2012, TU Darmstadt and FAU Erlangen-Nürnberg, 2012.

[9] Antony Phillip; Dr. Stuart Mitchell. A blending problem. `https://pythonhosted.org/PuLP/CaseStudies/a_blending_problem.html`, 2007.

[10] MOSEK.com. Getting started. `https://www.mosek.com/resources/getting-started/`.

[11] Dash Optimization. Xpress-mosel: user guide. *Englewood Cliffs, NJ*, 2002.

[12] GUROBI OPTIMIZATION. Software downloads and license center. `https://www.gurobi.com/downloads/`.

[13] R-project.org. What is r? `https://www.r-project.org/about.html`.

[14] Rdocumentation.org. Linear and integer programming. `https://www.rdocumentation.org/packages/lpSolve/versions/5.6.13.2/topics/lp`.

[15] Readthedocs.io. Modeling nonlinear problems. `https://pyomocontrib-simplemodel.readthedocs.io/en/latest/sodacan.html`.

[16] Readthedocs.io. Working with pyomo models. `https://pyomo.readthedocs.io/en/stable/working_models.html`.