Shu Lu *and* Quoc Tran-Dinh

Introduction to Optimization

From Linear Programming to Nonlinear Programming

October 8, 2019

STOR-UNC-Chapel Hill

Chapter 6

Software for Linear Programming

6.1 Overview

Computer software for linear programming is perhaps the most well developed area in optimization software, and is substantially reliable in practice. This chapter briefly presents an overview on computer software for LPs. We distinguish between two types of computer software.

- **LP solvers:** Roughly speaking, an LP solver is a computer program that implements one or many solution methods to solve LPs in a given form. There are at least two kinds of methods that are often implemented in an LP solver: simplex methods and interior-point methods. Nowadays, the list of LP solvers is getting longer and longer. For instance, CPLEX or `linprog` are LP solvers.
- **Optimization modeling languages:** Optimization modeling languages provide interfaces between model developers and optimization solvers. Modeling languages are "developed with the aim of allowing users to express LP and other optimization problems in a natural, algebraic form similar to the original mathematical expressions," as put in [1]. Data and models can be separated, making it easy to audit and adapt the models. The model developers do not have to arrange data into matrices or vectors, as this task will be done by the compiler. The compiler will translate the model into an acceptable format and pass it to a solver, which receives the problem and solves it. An optimization modeling software is often connected to various solvers ((either built-in or third-party solvers)) that handle different types of problems.

Modeling languages are good choices for model developers who deal with optimization problems of moderate to large-scale sizes and do not intend to write more codes than just the models. By using a modeling language in this course, we can focus on how to build the mathematical models for various types of problems, rather than spending too much effort on coding-specific issues. We do need to master the basics of a modeling language to be able to use it at an introductory level.

Let us start discussing some common LP solvers before using GAMS as a modeling language to illustrate the various applications. While many LP solvers are available, we focus on LP solvers that are readily accessible to most users.

6.2 LP solvers

We discuss Excel in Microsoft Office, `linprog` in Matlab, CPLEX, and some other LP solvers.

6.2.1 Excel

Let us start with Excel, a Microsoft's software program. The Solver add-in of Microsoft Excel is an easy-to-use tool and is widely used in business worldwide. We illustrate how to use this tool with a small example.

Problem description: A company produces two types of picture frames: each type-1 frame uses 2 hours of labor and 1 ounce of metal and brings $2.25 in profit, and each type-2 frame uses 1 hour of labor and 2 ounces of metal and brings $2.60 in profit. Each week 4000 labor hours and 5000 ounces of metal are available, and the company aims to maximize its weekly profit with an optimal production plan.

Problem modeling: To model the problem as a linear program, we use x_i to denote the number of type i picture frames produced weekly for $i = 1, 2$, and formulate the following LP:

$$\begin{cases} \max\limits_{x} & z = 2.25x_1 + 2.60x_2 & & \\ \text{s.t.} & 2x_1 + x_2 & \leq 4000, \\ & x_1 + 2x_2 & \leq 5000, \\ & x & \geq 0. \end{cases} \tag{6.1}$$

Solving this LP by Excel: When using Excel Solver for the first time, it is often necessary to activate it. The procedure depends on the version of Excel, and it often involves accessing the "Add-Ins" menu in Excel "Options" to select the "Solver

Add-In" to be active. Once activated, one can set up the problem in Excel as shown in Figure 6.1.

We enter data of the problem into the cells B5, C5, B7, C7, B8, C8, F7 and F8, and use cells B10 and C10 to store an arbitrarily chosen starting solution $(0,0)$. We use cell D5 to store the objective value, which is given by the formula `=SUMPRODUCT(B5:C5,$B$10:$C$10)`. The $ signs are to anchor locations of cells B10 and C10 when copying this formula to cells below. We use cells D7 and D8 for values of the left hand sides of the labor and metal constraints respectively, given by formulas `=SUMPRODUCT(B7:C7,$B$10:$C$10)` and `=SUMPRODUCT(B8:C8,$B$10:$C$10)` respectively. The $\leq$ signs in E7 and E8 are for display purposes and will be ignored by Excel Solver.

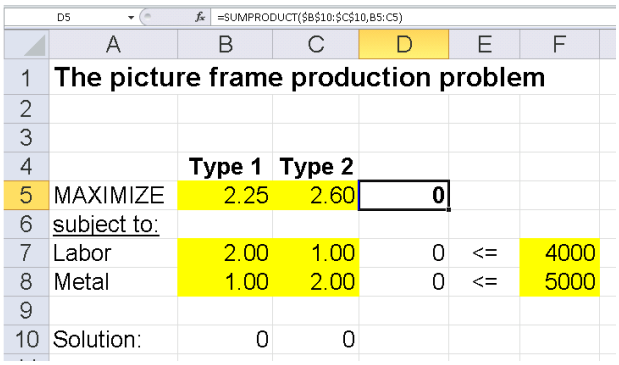

Fig. 6.1 Model and solve the **picture frame production problem** in Excel.

To start using the Excel Solver in Excel 2010, click **Solver** under the **Data** menu to open the Solver Parameters window.

- In the **Set Objective** box, enter the cell which contains the formula for computing the objective value (cell D5 for the example).
- Select **max** in the line below since we want to maximize the profit. For the **By Changing Variable Cells**, select the cells which contain the values of the decision variables (cells B10:C10 for the example).
- Next, add constraints one by one: select the cell that contains the left hand side formula as the **Cell Reference**, choose $<=$, $>=$, or $=$ based on the type of constraint, and enter the cell that contains the right hand side in the **Constraint** box.

- Following that, check the box of making unconstrained variables non-negative, and select Simplex LP as the solving method.
- Finally, click **Solve**, and the Solver Results box will tell us if Solver finds a solution. If Solver finds a solution, ensure that "Keep Solver Solution" is marked and click **OK**.

The optimal solutions for the decision variables will be placed in the Changing Cells.

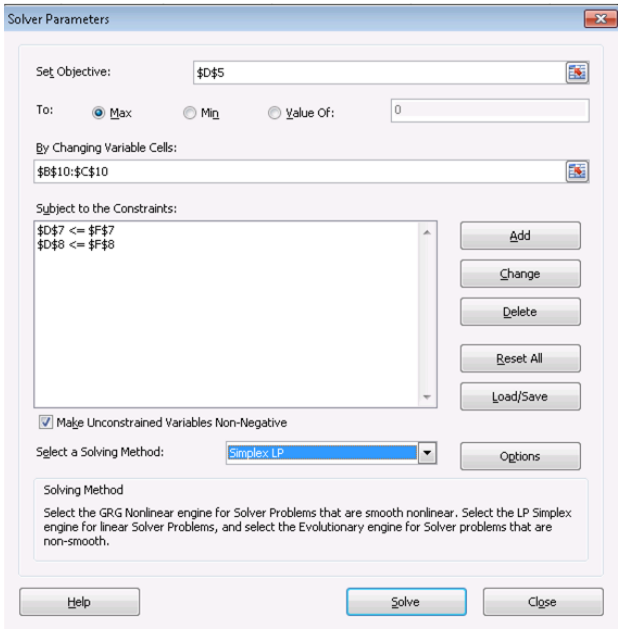

Fig. 6.2 Completed Excel Solver dialog box.

Final remarks: As we can see, the Excel Solver is easy and intuitive. With the wide availability of Excel, the Solver add-in is extensively used for simple optimization tasks. However, in general, it is not convenient to use Excel Solver for large-scale optimization models. It is hard to model complicated algebraic constraints using Excel Solver, and its computational performance is not as competitive as software specialized in optimization. Moreover, any dimensional or structural changes in the data would demand a major revision of the Solver parameters.

6.2.2 Matlab's LP solver

Matlab provides several routines for optimization in the Optimization Toolbox. It is often integrated in Matlab when we install it. More information about Matlab can be found at https://www.mathworks.com. The LP solver, called `linprog`, in the Optimization Toolbox of Matlab can solve relatively large-scale LPs. It accepts matrices and vectors as inputs. To see how we can input data of an LP, from the Command Window of Matlab, type

```
>> help linprog
```

for instructions on how to use this function. Below, we use `linprog` to solve the above picture frame example. We can use the following commands in the Command Window of Matlab:

```
>> c = [-2.25; -2.60];
>> A = [2 1; 1 2];
>> b = [4000;5000];
>> lb = zeros(2,1);
>> [x, z] = linprog(c, A, b, [], [], lb)
```

The first to the third lines enter the input data for c, A and b, respectively. The fourth line give a lower bound on x, which is 0. This imposes that $x \geq 0$. The last command calls `linprog` to minimize the linear objective function specified by the vector c, subject to linear constraints defined by the matrix A and the right hand side vector b and the lower bound lb on the variables. The empty matrices [] in the inputs indicate that there are no equality constraints. The outputs will be $x = (1000, 2000)^T$ with the optimal value $z = -7450$ as we can see below:

```
x =
   1.0e+03 *
   0.999999982545405
   2.000000004206820
z =
    -7.449999971664894e+03
```

We note that `linprog` also provides several options so that we can choose different methods, accuracy and monitor outputs to meet our expectation.

As illustrated by the above example, in order to use Matlab for solving LPs, one needs to arrange the input data (A, b, c) into matrices and vectors. For nonlinear optimization problems that involve nonlinear objective functions or constraints, one needs to define Matlab functions and pass them as function arguments in the inputs. For large-scale problems with complicated constraints, specifying the matrices and vectors can become a cumbersome task that makes it inconvenient to audit the codes or adapt the codes in response to changes in the model. However, this can be done by providing an interface on top of the solver to do this preprocessing step. Matlab provides several tools and routines to do such a task.

6.2.3 Other LP solvers

There are many other solvers for LPs. Below is a non-exhaustive list of LP solvers due to H. Mittelmann (see http://plato.asu.edu/ftp/lpsimp.html).

- **CPLEX:** CPLEX is a high-performance mathematical programming solver for linear programming, mixed integer programming, and quadratic programming. The name came from Simplex method implemented in C. This software program is currently owned by IBM, and is still active. CPLEX has a long history of development. It was founded by Robert E. Bixby and was sold commercially in 1988 by CPLEX Optimization Inc. More information about this solver can be found at http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html. CPLEX is available free of charge for academic use and for students.
- **Gurobi:** The Gurobi Optimizer is a new mathematical programming solver and is able to handle all major problem types, including linear programming. More information can be found here: http://www.gurobi.com. Gurobi is available free of charge for academic use and for students. Gurobi is named from its founders: Zonghao **Gu**, Edward **Ro**thberg and Robert **Bi**xby (also the father of CPLEX). Gurobi is very fast and can solve really big problems.
- **Mosek:** MOSEK is a tool for solving mathematical optimization problems such as LPs, quadratic programming, conic problems and mixed integer programming. MOSEK ApS was established in 1997 by Erling D. Andersen and Knud D. Andersen from Denmark. This software program is available at http://www.mosek.com and is free-of-charge for academic use.

- **Xpress:** XPRESS-Optimizer is a commercial optimization solver for many mathematical programming problems including linear programming (LP). More information about this package can be found at http://www.fico.com/en/products/fico-xpress-optimization-suite.
- **CLP:** CLP (Coin-or linear programming) is an open-source linear programming solver written in C++. More information can be found at https://projects.coin-or.org/Clp.
- **Google-GLOP:** This is also an LP solver developed by Google. See https://developers.google.com/optimization/lp/ for more detail.
- **SoPlex:** SoPlex is an optimization package for solving linear programs based on an advanced implementation of the primal and dual revised simplex algorithm. This software program is available at http://soplex.zib.de.
- **LP_SOLVE:** Lp_solve is a Mixed Integer Linear Programming (MILP) solver. It is an open-source package implemented in C. This package was originally developed by Michel Berkelaar at Eindhoven University of Technology. The code and information are available at http://lpsolve.sourceforge.net.
- **GLPK:** The GNU Linear Programming Kit (GLPK) is a software package intended for solving large-scale linear programs and other problems. It is an open-source package, and can be found at http://www.gnu.org/software/glpk/glpk.html. This package was developed by Andrew O. Makhorin, and remains active.

6.3 Algebraic modeling languages

AMLs (Algebraic modeling languages) are high-level computer programming languages for expressing and solving complex, large-scale mathematical optimization problems. The syntax adopted by an AML is often close to the natural language syntax used to describe a problem. There exist many optimization modeling languages including LINGO, AIMMS, AMPL, GAMS, OPL and Mosel. These software programs often support multiple optimization solvers including those discussed above. Below are more detailed information about some of these languages.

- **LINGO:** According to its website, LINGO is a comprehensive tool designed to build and solve linear, nonlinear (convex and nonconvex/global), quadratic, quadratically constrained, second-order cone, semidefinite, stochastic, and Integer optimization models. LINGO provides a integrated package that includes

a language for expressing optimization models, a full featured environment for building and editing problems, and a set of built-in solvers.

- **AIMMS:** According to the website https://aimms.com, the AIMMS (Publishing and Remote Optimization) platform is a collaboration platform in the field of optimization. It enables the use of AIMMS optimization applications through a web browser and offers a personal Enterprise Optimization App Store.
- **AMPL:** A Mathematical Programming Language (AMPL) is an algebraic modeling language to describe and solve highly complex, large-scale optimization problems. It first appeared in 1985 and was developed by Robert Fourer, David Gay, and Brian Kernighan at Bell Laboratories. It is still widely used in many areas of operations research. More information can be found at http://www.ampl.com.
- **GAMS:** GAMS (General Algebraic Modeling System) is a high-level modeling system for mathematical optimization. It is designed for modeling and solving linear, nonlinear, and mixed-integer optimization problems. More information can be founded at https://www.gams.com. We will use this software program extensively in this course.
- **OPL:** OPL was developed by IBM (http://www-01.ibm.com/software/commerce/optimization/modeling/index.htm It provides a natural mathematical description of optimization models. Its high-level syntax supports expressions needed to model and solve problems using both mathematical programming and constraint programming.
- **Mosel:** Similar to OPL, Mosel is a modeling language developed by a company called FICO (the owner of XPRESS). More information about this software can be found at http://www.fico.com/en/products/fico-xpress-optimization-suite.

Other computer software packages for solving optimization problems include AP-Monitor, ASCEND, OPTMODEL from SAS, Pyomo, and OptimJ, and so on.

6.4 GAMS: A General Algebraic Modeling System

GAMS was initially developed in the mid-1970s by Alex Meeraus and Jan Bisschop working at the World Bank. It later became a commercial product owned by the GAMS Development Corporation. From its website http://www.gams.com one can download free versions of GAMS.

In this section, we use the **picture frame example** described at the beginning of Section 6.2 to illustrate how to solve an optimization model in GAMS. We have two files, `picframe1.gms` and `picframe2.gms`, representing two different modeling styles that solve the same LP problem described in the problem formulation (6.1).

Each GAMS model is a collection of GAMS statements. Every statement is terminated with a semicolon ';' (experienced users can omit the semicolon when the next statement begins with a keyword). The most basic components in a GAMS model are:

- Variables
- Equations
- Model and solve statements.

Let us describe them in detail in the following subsections.

6.4.1 Variables: declaration and type assignment

When using GAMS to solve an optimization problem, we need to declare variables of the optimization problem in GAMS. In particular, we need to explicitly declare a variable to denote the quantity to be optimized. The last variable is called the **objective variable**.

Declaration: To declare a variable in GAMS, start with the keyword **variable** followed by the name for the variable. We can also add some descriptive text after the name. The following statements declare three variables with names `x1`, `x2` and `profit` respectively, where `profit` is the objective variable. The words between the quotation marks are descriptive texts that serve as comments. Experienced users can omit those quotation marks.

```
variable x1 "type 1 picture frames";
variable x2 "type 2 picture frames";
variable profit "total profit";
```

Alternatively, we can declare multiple variables in a single statement:

```
variables
     x1 "type 1 picture frames",
     x2 "type 1 picture frames",
```

```
profit "total profit";
```

Types of variables: Every GAMS variable is associated with a `type`. In this course we will use variables of the following five types.

- **Free** (the default type): a variable that can take any real value.
- **Positive**: a variable that can take any nonnegative value (GAMS uses the word "positive" to mean "nonnegative," so a positive variable in GAMS is allowed to be zero).
- **Negative**: a variable that can take any nonpositive value (a negative variable in GAMS is allowed to be zero).
- **Binary**: a variable that is either 0 or 1.
- **Integer**: a variable that can only take integer values between specified bounds (the bounds are 0 and 100 by default and can be changed).

We can declare the variables first and then assign types to them in a separate statement:

```
positive variables x1, x2;
```

We can also declare variables and assign their type in one statement. Note that the objective variable **must** be a free variable in a GAMS optimization model.

6.4.2 Equations: declaration and definition

To solve an optimization problem in GAMS, we need to specify the constraints and the objective function by **equations**. The keyword **equation** has a broad meaning in GAMS; it can mean either an equality or an inequality (see the example below). A GAMS equation defined with a common name can also stand for a family of equations of the same structure.

A GAMS equation needs to be declared first, and then defined in a separate statement. To declare an equation is to give it a name. To define an equation is to specify the relations between variables in this equation.

Declaration: To declare an equation, start with the keyword **equation** followed by the name for the equation. As with the declaration of variables, we can add some optional descriptive text after the name. The following statement declares three equations named `obj`, `labor` and `metal`.

```
equations
```

```
obj    "max total profit",
labor  "labor hours",
metal  "metal in oz";
```

Define equations: To define an equation, start with the name of the equation declared in a preceding statement, put two dots '..' after the name, write down the left-hand-side expression and the right-hand-side expression, and put a relational operator between the two expressions. A relational operator can be one of the following three types:

- **=g= Greater than**: LHS (the left-hand side) must be greater than or equal to RHS (the right-hand side).
- **=e= Equality**: RHS must equal LHS.
- **=l= Less than**: LHS must be less than or equal to RHS.

Keep in mind that we **must** use one of the above operators in defining GAMS equations instead of the usual symbols $\leq$, $=$, and $\geq$ in mathematical formulations. It is fine to use upper case letters (=G=, =E=, and =L=).

The following three statements define the three equations declared above, with the equation `obj` specifying the objective function, and equations `labor` and `metal` specifying the two constraints.

```
obj.. profit =e= 2.25*x1 + 2.60*x2;
labor.. 2*x1 + x2 =l= 4000;
metal.. 1*x1 +2*x2 =l= 5000;
```

6.4.3 Model and solve statements

The model statement is used to collect equations into groups and to label them so that they can be solved (advanced users may create several models in one GAMS file).

Declaration: To declare a model, start with the keyword **model** followed by the name for the model, followed by a list of equation names enclosed in slashes. For example, the statement below declares a model named `picframe`, which collects the three equations declared before.

```
model picframe /obj,labor,metal/;
```

In fact, if all previously defined equations are to be included in a model, we can use `/all/` in place of the explicit list. That is

```
model picframe /all/;
```

Call solver: Once a model has been declared and assigned equations, we are ready to call the solver. This is done with a solve statement. A "solve statement" starts with the keyword **solve** followed by the name of the model to be solved, then another keyword **using** followed by the model type, then the keyword **minimizing** or **maximizing**, and ends with the name of the objective variable. That is

```
solve Model_Name using Model_Type maximizing [minimizing] Objective_variable;
```

Below is the solve statement in the example:

```
solve picframe using lp maximizing profit;
```

where `picframe` is the name of the model just declared, "lp" is the model type, and `profit` is the name of the objective variable declared earlier.

Types of models: The model type "lp" stands for linear programming. GAMS can be used to solve various types of models, including

- **lp** (or LP): for linear programming;
- **qcp** (or QCP): for quadratic constraint programming;
- **nlp** (or NLP): for nonlinear programming;
- **mip** (or MIP): for mixed integer programming, etc.

In summary, we obtain the following gams model, and save it into a `*.gms` file called `picframe1.gms`:

```
*picframe1.gms: This is a GAMS model for example (1) above.
*Variable declaration
variables
    x1 "type 1 picture frames",
    x2 "type 2 picture frames",
    profit "total profit";
*Assign type of variables
positive variables x1, x2;
*Equation declaration
equations
```

```
     obj    "max total profit",
     labor  "labor hours",
     metal  "metal in oz";
*Equation definition
  obj..profit =e= 2.25*x1 + 2.60*x2;
  labor..2*x1 + x2 =l= 4000;
  metal..1*x1 +2*x2 =l= 5000;
*Model and solve statements
model picframe /all/;
solve picframe using lp maximizing profit;
```

6.4.4 GAMS outputs

To compile and execute a GAMS file, select "Run" in "File" menu in GAMS-IDE, or click the "Run" button on the toolbar. In response to that command, GAMS will compile the file and call a solver to solve the model if it does not find any syntax errors. GAMS has integrated many solvers developed by various people in industry and academia. We can see the list of solvers in the "File $\rightarrow$ Options $\rightarrow$ Solvers" window. Each solver handles different types of models, and each type of models has a default solver. For example, the system default solver to solving LP models is CPLEX.

During the compilation / execution and solution phases, a process window will pop up to show the progress of a GAMS job. If the file contains syntax errors, error messages will be shown in red with an indication where the error occurred.

After closing the process window, we will find the "`lst`" file opened automatically in GAMS-IDE. The `*.lst` file contains output produced from the GAMS run. It is saved in the currently active **project directory**. GAMS-IDE has a default project directory under which all output files will be saved. We can reset the directory by creating a new project at a convenient location. Information we can find in the `*.lst` file include:

(a) Error messages (if there is any error).

(b) The solution report:

- Solver status: the desired status is 1: normal completion; any other solver status indicates something wrong with the solution process.

- Model status: for an LP model there are three possible model statuses: 1 - optimal, 3 - unbounded, 4 - infeasible.
- Objective value: the optimal value found by the solver.
- The **levels** of variables: This indicates the values of the **optimal solution** found by the solver. A single dot '.' in the `*.lst` file represents a zero value.
- The "marginal" column for equations: This indicates the values of dual variables.

6.4.5 Language items

This subsection collects small tips for using the GAMS language correctly. Detailed instructions and discussion can be found in Section 3.4 of `GAMS User Guide`, which is accessible from the "Help" menu in GAMS-IDE.

- **Names:** Names given to GAMS variables, equations, models, etc are all called **identifiers**. There are certain rules one needs to follow in picking an identifier. For example, it must start with a letter followed by more letters or digits. It is worth mentioning that GAMS is case-insensitive. Hence, `x1` and `X1` can be used interchangeably in GAMS.
- **Keywords:** Keywords (also called reserved words) in GAMS are words with predefined meanings. It is not permitted to use a keyword as an identifier. For example, we cannot name a variable just as "variable" or "VARIABLE", because the words "variable" and "VARIABLE" are reserved as keywords by GAMS. The complete list of reserved words is in table 3.3 of the Guide.
- **Separation:** As we know semicolons are used to separate GAMS statements. Inside a statement, one can use commas as delimiters to separate names of different variables or equations.
- **Comments:** Finally, we can add a line of comments by putting the symbol "*" as the first character of that line. We can also create a paragraph of comments between the options `$ontext` and `$offtext`. Comments are ignored when the file is compiled.

6.4.6 Set declaration

An important common feature of optimization modeling languages is the use of sets to collect indices. This feature allows the model to be succinctly stated and easily read, making it convenient to model large-scale problems.

To declare a set, start with the keyword **set** or **SET** followed by the name of the set, followed by members (also called elements) of the set between two slashes. The following statement in `picframe2.gms` declares a set named `I` that consists of two elements "type-1" and "type-2".

```
set I /type-1, type-2/;
```

Note that elements of a set are labels but not numbers. To emphasize this, we can put the labels in quotes in the above statement:

```
set I /'type-1', 'type-2'/;
```

We note that we can define multiple sets at the same time. For example,

```
SETS I "variables" /type-1, type-2/, J "constraints" /labor, metal/;
```

6.4.7 Scalars and parameters

To declare scalars and parameters in GAMS is to give names to the various numbers that appear in a problem formulation. By using scalars and parameters instead of specific numbers in the definitions of GAMS equations, we can separate data from the equations, making the model easier to audit, adapt and manage.

(a) Scalars: Scalars are used for single data entries that are not associated with any set. We can declare a scalar by using the keyword **scalar** followed by the name of the scalar, followed by the value of the scalar between two slashes. As with other GAMS entities, we can put optional descriptive text after the name of the scalar. The following statement in `picframe2.gms` declares two scalars, named `NumLabor` and `NumMetal` respectively, with values 4000 and 5000 respectively.

```
scalar NumLabor "Number of labor hours available" /4000/,
       NumMetal "Metal (oz) available" /5000/;
```

(b) Parameters: Parameters are used for data over index sets. The set over which a parameter is indexed is called the **domain** of the parameter. To declare a parameter, start with the keyword **parameter** followed by the name of the parameter, put the domain of the parameter between parentheses, then assign values of the parameter for each element in the domain between two slashes. The following statement declares parameters named `sellingPrice`, `laborUse` and `metalUse` over the set `I`, and assigns their values. For example, `sellingPrice("type-1")` takes the value 2.25 and `metalUse("type-2")` takes the value 2.

```
parameters sellingPrice(I) /type-1 2.25, type-2 2.60/,
           laborUse(I) /type-1 2, type-2 1/,
           metalUse(I) /type-1 1, type-2 2/;
```

6.4.8 Variables over sets

One can declare a variable over a set, by putting the name of the set in parentheses after the name of the variable in the statement that declares the variable. The set is called the domain of that variable. To declare a variable over a set is to name a group of variables using elements of the set as indices. The following statement from `picframe2.gms` declares a variable named `x` over the set `I`. With this declaration, we have two variables `x("type-1")` and `x("type-2")` to be used in equations.

```
positive variables x(I) "number of picture frames";
```

If we declare the variable `x` first and then assign type to it in a separate statement, then we should not put its domain in the second statement. Example:

```
variables x(I) "number of picture frames";
positive variable x;
```

6.4.9 Equations with scalars, parameters and variables over sets

We can use scalars and parameters in defining equations. Recall the equations defined in `picframe1.gms`:

```
obj.. profit =e= 2.25*x1 + 2.60*x2;
labor.. 2*x1 + x2 =l= 4000;
metal.. 1*x1 +2*x2 =l= 5000;
```

After replacing the numbers by their names and replacing variables `x1` and `x2` by `x("type-1")` and `x("type-2")`, we obtain the following definitions for the equations:

```
obj.. profit =e= sellingPrice("type-1")*x("type-1") + sellingPrice("type-2")*x("type-2");
labor.. laborUsed("type-1")*x("type-1") + laborUsed("type-2")*x("type-2") =l= NumLabor;
metal.. metalUsed("type-1")*x("type-1") +  metalUsed("type-2")*x("type-2") =l= NumMetal;
```

The summations in above equations can be written compactly using the GAMS summation notation. The basic format is `Sum(index of summation, summand)`. With this notation the above equations can be written as

```
obj.. profit =e=   sum(I, sellingPrice(I)*x(I));
labor.. sum(I, laborUsed(I)*x(I)) =l= NumLabor;
metal.. sum(I, metalUsed(I)*x(I)) =l= NumMetal;
```

In the above statements, the set `I` is the index of summation.
The expression `sum(I, sellingPrice(I)*x(I))` gives the sum of `sellingPrice(I)*x(I)` over all indices in `I`, and is therefore an equivalent way to express the sum

```
sellingPrice("type-1")*x("type-1") + sellingPrice("type-2")*x("type-2")
```

Clearly, if the number of elements in `I` is large, it is much more convenient to use the command `sum`.

A complete script of the file `picframe3.gms` in GAMS is given below, including comments.

```
$ontext
  A company produces two picture frame types.
  Each Type 1 frame uses 2 hr labor and 1 oz metal, and brings $2.25 of profit.
  Each Type 2 frame uses 1 hr labor and 2 oz metal, and brings $2.60 of profit
  There are 4000 hours labor and 5000 oz metal available each week. How many
  frames of each type should this company produce to maximize weekly
  profit?
$offtext
*set declaration
set I /type-1, type-2/;
*scalars and parameters
scalar
    NumLabor /4000/,
    NumMetal /5000/;
parameters
    price(I)         /type-1 2.25, type-2 2.60/,
    laborUse(I)  /type-1 2,       type-2 1/,
    metalUse(I) /type-1 1,       type-2 2/;
*variable declaration and type assignment
free variable
    profit "total profit";
positive variables
    x(I) "number of picture frames";
*equation declaration
equations
    obj    "max total profit",
    labor  "labor hours",
```

```
    metal  "metal in oz";
*equation definition
    obj.. profit =e=   sum(I, price(I)*x(I));
    labor.. sum(I, laborUse(I)*x(I)) =l= NumLabor;
    metal.. sum(I, metalUse(I)*x(I)) =l= NumMetal;
model picframe /all/;
solve picframe using lp maximizing profit;
```

If we call GAMS to solve this problem, we will obtain an output file called `picframe3.lst` as below.

```
  S O L V E      S U M M A R Y
     MODEL   picframe            OBJECTIVE  profit
     TYPE    LP                  DIRECTION  MAXIMIZE
     SOLVER  CPLEX               FROM LINE  40
**** SOLVER STATUS     1 Normal Completion
**** MODEL STATUS      1 Optimal
**** OBJECTIVE VALUE             7450.0000
 RESOURCE USAGE, LIMIT          0.016      1000.000
 ITERATION COUNT, LIMIT         2    2000000000
IBM ILOG CPLEX   24.4.6 r52609 Released Jun 26, 2015 WEI x86 64bit/MS Windows
Cplex 12.6.2.0
Space for names approximately 0.00 Mb
Use option 'names no' to turn use of names off
LP status(1): optimal
Cplex Time: 0.00sec (det. 0.01 ticks)
Optimal solution found.
Objective :        7450.000000
                      LOWER     LEVEL     UPPER    MARGINAL
---- EQU obj             .         .         .        1.000
---- EQU labor          -INF   4000.000  4000.000     0.633
---- EQU metal          -INF   5000.000  5000.000     0.983
  obj  max total profit
  labor  labor hours
  metal  metal in oz
                      LOWER     LEVEL     UPPER    MARGINAL
---- VAR profit         -INF   7450.000     +INF       .
  profit  total profit
---- VAR x  number of picture frames
          LOWER     LEVEL     UPPER    MARGINAL
type-1      .     1000.000     +INF       .
```

```
type-2      .     2000.000     +INF       .
**** REPORT SUMMARY :        0     NONOPT
                             0 INFEASIBLE
                             0  UNBOUNDED
EXECUTION TIME       =        0.000 SECONDS      2 MB  24.4.6 r52609 WEX-WEI
USER: GAMS Development Corporation, Washington, DC   G871201/0000CA-ANY
      Free Demo,  202-342-0180,  sales@gams.com,  www.gams.com   DC0000
```

6.5 Exercises

Exercise 6.1. Young MBA Erica Cudahy plans to invest up to $1000 to a financial market. She can invest her money in stocks and loans. Each dollar invested in stocks yields 10 cents of profit, and each dollar invested in a loan yields 15 cents of profit. At least 30% of all money invested must be in stocks, and at least $400 must be in loans. Formulate an LP problem to maximize total profit earned from Erica's investment.

Create a GAMS file named `Erica.gms`. In the GAMS code, declare two positive variables, *stock* and *loan*, to denote the amount of money to put in stock and loan respectively. Also declare a free variable called *profit* to denote the total profit. Model and solve as an LP.

Exercise 6.2. Cuppa Coffee Company mixes specialty coffee blends to sell to SmartBux, a small chain of coffee shops. The ingredients for their specialty coffee are the following beans:

Bean Type	Cost per pound	Available amount (pounds)
Columbian	$1.00	550
Brazilian	$0.85	450
Sumatran	$1.55	650

Cuppa's products are:

- **Robust Joe** must consist of 60%–75% Sumatran beans and at least 10% Columbian beans. Each pound of Robust Joe can be sold to SmartBux for $4.25.
- **Light Joe** must consist of 50%–60% Brazilian beans and no more than 20% Sumatran beans. Each pound of Light Joe can be sold to SmartBux for $3.95.

Formulate an LP problem to maximize total profit. To check the correctness of your solution, we provide the optimal value, which is the maximum profit, to be $4902.5.

Create a GAMS file named `cuppacoffee.gms`. In the GAMS code, declare

```
SETS I /Columbian, Brazilian, Sumatran/, J /Robust, Light/;
```

as two sets of indices, and then declare a variable `x` over `(I,J)`. Declare the bean supply, purchase prices and coffee prices as parameters over `I` or `J`. For simplicity, you can use numbers for percentages in the equations (without declaring them as scalars). Example:

```
equations  RS1 "Robust has at least 60 percent of sumatran";
    RS1.. x("Sumatran","Robust")  =g= 0.6* sum(I, x(I,"Robust"));
```

Exercise 6.3. James Beerd bakes two types of cakes: cheesecakes and black forest cakes. During any month, he can bake at most 65 cakes in total. The costs per cake and the demands for cakes, which must be met in time, are given in the following table.

	Month 1		Month 2		Month 3	
Item	Demand	Cost/cake($)	Demand	Cost/cake($)	Demand	Cost/cake($)
Cheesecake	40	3.00	30	3.40	20	3.80
Black Forest	20	2.50	30	2.80	10	3.40

We assume that cakes baked during a month can be used to meet demand for this month. At the end of each month (after all cakes have been baked and the current month's demand has been satisfied), a holding cost of 50 cents per cheesecake and 40 cents per black forest cake is incurred for cakes left in inventory. Those cakes can be used to satisfy future demand.

Formulate an LP problem to minimize the total cost of meeting the next three months' demands. Solve it using GAMS. Create a GAMS file named `cake.gms`. Declare two sets in the GAMS code:

```
SETS i /cc, bf/, t /1*3/;
```

Declare the holding costs at a parameter over `i`, and the demands and production costs as tables over `(i,t)`. Then declare positive variables `x(i,t)` and `a(i,t)`, to represent numbers of each type of cakes to make each month, and numbers of each type of cakes left in inventory at the end of each month respectively. To check the correctness of your solution: the optimal value is $464.5.

References

1. Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.