The GNU Linear Programming Kit, Part 1: Introduction to linear optimization

Find the best solutions to complex numeric problems

Rodrigo Ceron (rceron@br.**ibm**.com), Staff Software Engineer, **IBM**, Software Group

Summary: The **GNU** Linear Programming Kit is a powerful, proven tool for solving numeric problems with multiple constraints. This article introduces GLPK, the glpsol client utility, and the **GNU MathProg** language to solve the problem of optimizing the operations for Giapetto's Woodcarving, Inc., a fictional toy manufacturer.

Date: 08 Aug 2006
Level: Intermediate
Activity: 12281 views
Comments: ●

Introduction

> "Linear programming is a tool for solving optimization problems. In 1947, George Dantzig developed an efficient method, the simplex algorithm, for solving linear programming problems. Since the development of the simplex algorithm, linear programming has been used to solve optimization problems in industries as diverse as banking, education, forestry, petroleum, and trucking. In a survey of Fortune 500 firms, 85% of the respondents said they had used linear programming."
>
> From *Operations Research: Applications and Algorithms, 4th Edition*, by Wayne L. Winston (Thomson, 2004); see Resources below for a link.

Many tools are available to solve linear programming problems. The proprietary tools are well known, but many members of the open source community may not know about the free GLPK tool.

The first in a series of three articles that show GLPK's capabilities and usage, this article briefly describes GLPK and then demonstrates and applies the **GNU MathProg** Language in GLPK.

If you are just starting with operations research theory and want to learn how to model and solve linear problems, this article is a good guide.

The **GNU** Linear Programming Kit

The **GNU** Linear Programming Kit (GLPK) is a library of routines that use well-known operations research algorithms to solve linear problems. The routines implement the simplex, branch and bound, primal-dual interior point, and many other algorithms. Check the GLPK manual included with the GLPK download to find out more. (To download the GLPK, see the Resources section for a link to the GLPK page on **gnu**.org.)

GLPK is not a program -- it can't be run and has no `main()` function. Instead, clients feed the problem data to the algorithmic routines through the GLPK API and receive results back. GLPK has a default client, the glpsol program, that interfaces with this API. Usually, a program like glpsol is called a *solver* rather than a client, so you'll see this nomenclature from here forward.

The **GNU MathProg** modeling language

The **GNU MathProg** modeling language is nice and simple for declaring linear problems. In general, a problem declaration consists of:

- Problem decision variables.
- An objective (target) function. Note that *objective* is a noun, not an adjective. The name is standard in operations research theory.
- Problem constraints.
- Problem parameters (data).

Let's start with a simple two-variable example: Giapetto's Woodcarving, Inc.

Giapetto's Woodcarving Inc.

This problem is from *Operations Research*:

Giapetto's Woodcarving Inc. manufactures two types of wooden toys: soldiers and trains. A soldier sells for $27 and uses $10 worth of raw materials. Each soldier that is manufactured increases Giapetto's variable labor and overhead costs by $14. A train sells for $21 and uses $9 worth of raw materials. Each train built increases Giapetto's variable labor and overhead costs by $10. The manufacture of wooden soldiers and trains requires two types of skilled labor: carpentry and finishing. A soldier requires 2 hours of finishing labor and 1 hour of carpentry labor. A train requires 1 hour of finishing and 1 hour of carpentry labor. Each week, Giapetto can obtain all the needed raw material but only 100 finishing hours and 80 carpentry hours. Demand for trains is unlimited, but at most 40 soldier are bought each week. Giapetto wants to maximize weekly profits (revenues - costs).

To summarize the important information and assumptions about this problem:

1. There are two types of wooden toys: soldiers and trains.
2. A soldier sells for $27, uses $10 worth of raw materials, and increases variable labor and overhead costs by $14.
3. A train sells for $21, uses $9 worth of raw materials, and increases variable labor and overhead costs by $10.
4. A soldier requires 2 hours of finishing labor and 1 hour of carpentry labor.
5. A train requires 1 hour of finishing labor and 1 hour of carpentry labor.
6. At most, 100 finishing hours and 80 carpentry hours are available weekly.
7. The weekly demand for trains is unlimited, while, at most, 40 soldiers will be sold.

The goal is to find the numbers of soldiers and trains that will maximize the weekly profit.

Modeling

To model a linear problem, the decision variables are established first, since they will change with each iteration of the simplex algorithm and determine the value of the objective function and, hence, the optimal solution. In Giapetto's shop, the objective function is the profit, which is a function of the amount of soldiers and trains produced each week. Therefore, the two decision variables in this problem are:

- x_1: Number of soldiers produced each week
- x_2: Number of trains produced each week

Once the decision variables are known, the objective function of this problem is simply the revenue minus the costs for each toy, as a function of x_1 and x_2.

$$z = (27 - 10 - 14)x_1 + (21 - 9 - 10)x_2 = 3x_1 + 2x_2 \tag{1}$$

Note that the profit depends linearly on x_1 and x_2 -- this is a linear problem.

It may seem at first glance that the profit can be maximized by simply increasing x_1 and x_2. Well, if life were that easy, let's start manufacturing trains and soldiers and move to the Caribbean! Unfortunately, there are restrictions that limit the decision variables that may be selected (or else the model is very likely to be wrong).

Recall the assumptions made for this problem. The first three determined the decision variables and the objective function. The fourth and sixth assumption say that finishing the soldiers requires time for carpentry and finishing. The limitation here is that Giapetto doesn't have infinite carpentry and finishing hours. That's a constraint! Let's analyze it to clarify.

One soldier requires 2 hours of finishing labor, and Giapetto has at most 100 hours of finishing labor per week, so he can't produce more than 50 soldiers per week. Similarly, the carpentry hours constraint makes it impossible to produce more than 80 soldiers weekly. Note here that the first constraint is stricter than the second. The first constraint is effectively a subset of the second, thus the second constraint is redundant.

The previous paragraph shows how to model optimization problems, but it's an incomplete analysis because all the necessary variables were not considered. It's not the complete solution of the Giapetto problem. So how should the problem be approached?

Start by analyzing the limiting factors first in order to find the constraints. First, what constrains the finishing hours? Since both soldiers and trains require finishing time, both need to be taken into account. Suppose that 10 soldiers and 20 trains were built. The amount of finishing hours needed for that would be 10 times 2 hours (for soldiers) plus 20 times 1 hour (for trains), for a total of 40 hours of finishing labor. The general constraint in terms of the decision variables is:

$$2x_1 + x_2 \leq 100 \tag{2}$$

There are many (x_1, x_2) pairs that satisfy this inequality, so this does not determine the best combination for Giapetto's shop.

Now that the constraint for the finishing hours is ready, the carpentry hours constraint is found in the same way to be:

$$x_1 + x_2 \leq 80 \tag{3}$$

Great! There's only one more constraint for this problem. Remember the weekly demand for soldiers? According to the problem description, there can be at most 40 soldiers produced each week:

$$x_1 \leq 40 \tag{4}$$

The demand for trains is unlimited, so no constraint can be written for it. The model is finished and consists of the equations:

$$max\ \ z = 3x_1 + 2x_2\ \ (objective\ function) \tag{5}$$

$$2x_1 + x_2 \leq 100\ \ (finishing\ constraint) \tag{6}$$

$$x_1 + x_2 \leq 80\ \ (carpentry\ \ constraint) \tag{7}$$

$$x_1 \leq 40\ \ (demand\ for\ soldiers) \tag{8}$$

$$x_1 \geq 0,\ \ x_2 \geq 0\ \ (sign\ contraints) \tag{9}$$

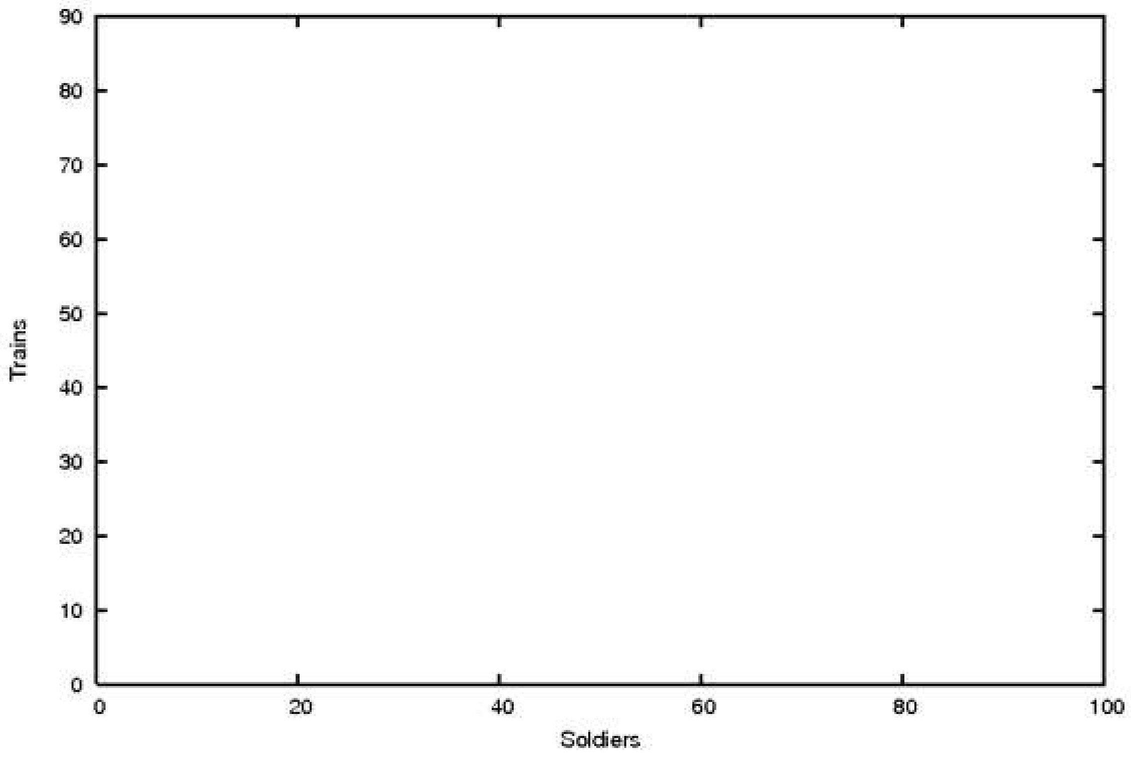
Note the last constraint. It ensures that the values of the decision variables will always be positive. The problem does not state this explicitly, but it's still important (and obvious).

Now GLPK can solve the model (since GLPK is good at solving linear optimization problems).

A little bit of theory

Let's check the problem's solution space. With two decision variables, it has two dimensions.

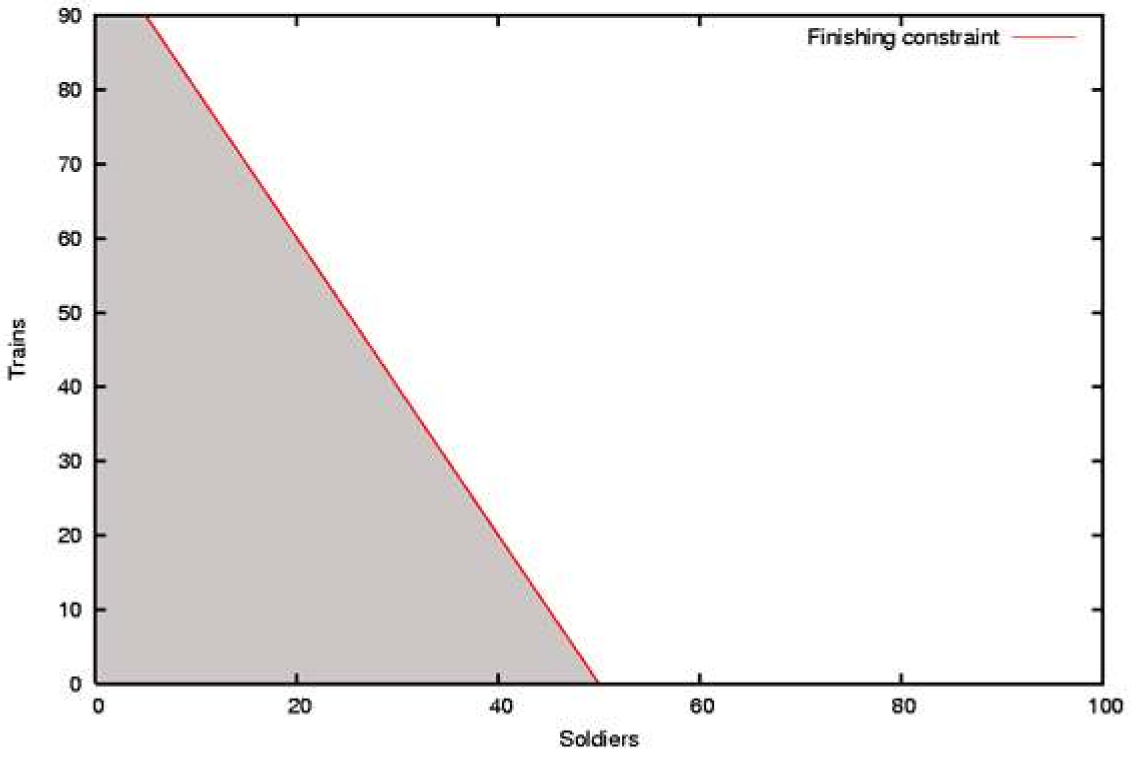
Figure 1. Giapetto's unbounded universe



The (x_1, x_2) solutions outside the first quadrant (where all values are positive) have already been discarded. Note, however, that this solution space is still infinite (that *would* be a situation in which I'd move to the Caribbean!)

As the constraints were written, this unlimited solution space gained boundaries. With inequality 6, above, the result is more interesting.

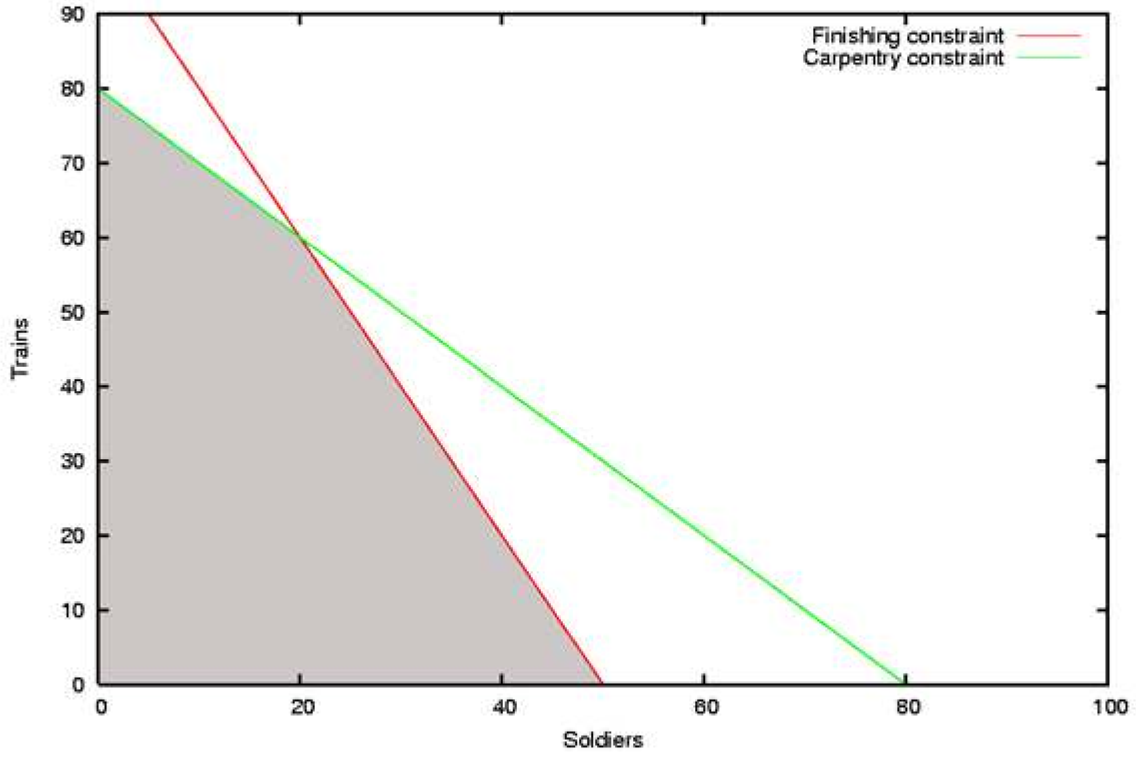
Figure 2. Giapetto's universe considering the finishing constraint



The solution space contains all the possible (x_1, x_2) solutions in the first quadrant that satisfy the finishing hours constraint.

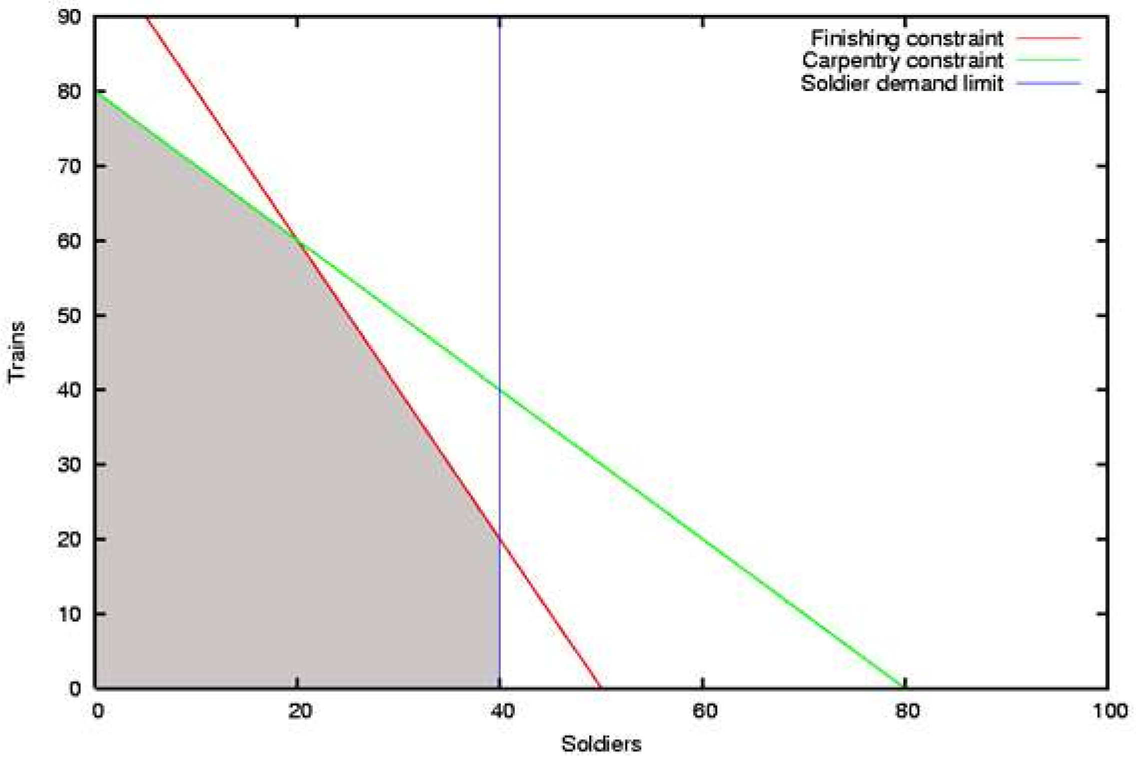
After inequality 7, the result set shrinks.

Figure 3. Giapetto's universe considering the finishing and carpentry constraints



Note that the solution space is smaller. This means that even fewer (x_1, x_2) solutions are in it. After inequality 8, the result is even smaller.

Figure 4. Giapetto's feasible region



The solution space gets smaller still. The solution space that satisfies all the constraints is called the *feasible region*. Figure 4 shows the feasible region for Giapetto's shop. Any (x_1, x_2) pair that falls into that region is a potential solution to the problem.

The question now is: which one maximizes Giapetto's profit?

Using GLPK to solve the model

GLPK is an excellent tool to solve that question. The mathematical formulation of Giapetto's problem needs to be written with the **GNU MathProg** language. The key items to declare are:

- The decision variables
- The objective function
- The constraints
- The problem data set

The following code shows how to solve Giapetto's problem with **MathProg**. The line numbers in this code are not part of the code itself. They have been added only for the sake of making references to the code.

Listing 1. First solution to Giapetto's problem: giapetto.sol

```
 1  #
 2  # Giapetto's problem
 3  #
 4  # This finds the optimal solution for maximizing Giapetto's profit
 5  #
 6
 7  /* Decision variables */
 8  var x1 >=0;  /* soldier */
 9  var x2 >=0;  /* train */
```

```
10
11  /* Objective function */
12  maximize z: 3*x1 + 2*x2;
13
14  /* Constraints */
15  s.t. Finishing : 2*x1 + x2 <= 100;
16  s.t. Carpentry : x1 + x2 <= 80;
17  s.t. Demand    : x1 <= 40;
18
19  end;
```

Lines 1 through 5 are comments. `#` anywhere on a line begins a comment to the end of the line. C-style comments can also be used, as shown on line 7. They even work in the middle of a declaration.

The first **MathProg** step is to declare the decision variables. Lines 8 and 9 declare x_1 and x_2. A decision variable declaration begins with the keyword `var`. To simplify sign constraints (check inequality 9), **MathProg** allows a `>= 0` constraint in the decision variable declaration, as seen on lines 8 and 9. Every sentence in **GNU MathProg** must end with a semicolon (`;`). Recall that x_1 represents `soldier` numbers and x_2 represents `train` numbers. These variables could have been called `soldiers` and `trains`, but that would confuse the mathematicians in the audience.

Line 12 declares the target (objective) function. Linear problems can be either maximized or minimized. Remember, Giapetto's mathematical model is a maximization problem, so the keyword `maximize` is appropriate instead of the opposite keyword, `minimize`. The objective function is named `z` and equals $3x_1 + 2x_2$. Note that:

- The colon (`:`) character separates the name of the objective function and its definition.
- The asterisk (`*`) character denotes multiplication and, similarly, the plus (`+`), minus (`-`), and forward slash (`/`) characters denote addition, subtraction, and division as you'd expect.

Lines 15, 16, and 17 define the constraints. Although `s.t.` is not required at the beginning of the line to declare a constraint, it improves the readability of the code.

The three Giapetto constraints have been labeled *Finishing*, *Carpentry*, and *Demand*. Each of them is declared as in the mathematical model. The symbols `<=` and `>=` express the inequalities. Don't forget the `;` at the end of each declaration.

Every **GNU MathProg** file must end with `end;`, as seen on line 19.

Now, glpsol can use this file as input. But wait a minute; where's the data section of this problem? Well, this problem is so simple that the problem data is directly included in the objective function and constraints declarations as the coefficients of the decision variables in the declarations. For example, in the objective function, the coefficients `3` and `1` are part of the problem's data set. When I rewrite this problem using a data set, it will become clear how it works; for now, don't worry about it.

It's good practice to use the `.mod` extension for **MathProg** input files and redirect the solution to a file with the extension `.sol`. This is not a requirement -- you can use any file name and extension you like. Giapetto's **MathProg** file for this example will be `giapetto.mod`, and the output will be in `giapetto.sol`. Now, run `glpsol` in your favorite console:

```
glpsol -m giapetto.mod -o giapetto.sol
```

This command line uses two glpsol options:

- The `-m` option tells glpsol that the input is written in **GNU MathProg**.
- The `-o` option tells glpsol to send its output to `giapetto.sol`.

The solution report will be in `giapetto.sol`, but some information about the time and memory GLPK consumed is shown on the system's standard output:

Listing 2. Output from glpsol

```
ceron@curly ~ $ glpsol -m giapetto.mod -o giapetto.sol
Reading model section from giapetto.real.mod...
19 lines were read
Generating z...
Generating Finishing...
Generating Carpentry...
Generating Demand...
Model has been successfully generated
lpx_simplex: original LP has 4 rows, 2 columns, 7 non-zeros
lpx_simplex: presolved LP has 2 rows, 2 columns, 4 non-zeros
lpx_adv_basis: size of triangular part = 2
*     0:   objval =   0.000000000e+00   infeas =   0.000000000e+00 (0)
*     2:   objval =   1.400000000e+02   infeas =   0.000000000e+00 (0)
OPTIMAL SOLUTION FOUND
Time used:   0.0 secs
Memory used: 0.1M (151326 bytes)
lpx_print_sol: writing LP problem solution to `giapetto.sol'...
```

The report shows that glpsol reads the model, calls a GLPK API function to generate the objective function, then calls another API function to generate the constraints. After the model has been generated, glpsol explains briefly how the problem was handled internally by GLPK. At the end, there's information about the solution and the resources used by GLPK to solve it, and the solution is noted to be optimal.

Great, but what are the actual optimal values for the decision variables? They are in the `giapetto.sol` file:

Listing 3. The solution to Giapetto's problem: giapetto.sol

```
Problem:    giapetto
Rows:       4
Columns:    2
Non-zeros:  7
Status:     OPTIMAL
Objective:  z = 180 (MAXimum)

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 z            B            180
     2 Finishing    NU           100                         100             1
     3 Carpentry    NU            80                          80             1
     4 Demand       B             20                          40

   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 x1           B             20             0
     2 x2           B             60             0

Karush-Kuhn-Tucker optimality conditions:

KKT.PE: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

KKT.PB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

KKT.DE: max.abs.err. = 0.00e+00 on column 0
        max.rel.err. = 0.00e+00 on column 0
        High quality

KKT.DB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

End of output
```

The solution is divided into four sections:

- Information about the problem and the optimal value of the objective function
- Precise information about the status of the objective function and about the constraints
- Precise information about the optimal values for the decision variables
- Information about the optimality conditions, if any

For this particular problem, we see that the solution is `OPTIMAL` and that Giapetto's maximum weekly profit is $180.

The Finishing constraint's status is `NU` (the *St* column). `NU` means that there's a non-basic variable on the upper bound for that constraint. If you know some operation research theory, build the simplex tableau and check it out. If you don't, here's a a brief practical explanation.

Whenever a constraint reaches its upper or lower boundary, it's called a *bounded constraint*. A bounded constraint prevents the objective function from reaching a better value. Think of it as a volume knob, for example, that can't be turned any further. When that occurs, glpsol shows the status of the constraint as either `NU` or `NL` (for upper and lower boundary respectively), and it also shows the value of the *marginal*, also known as the *shadow price*. The marginal is the value by which the objective function would improve if the constraint were relaxed by one unit (if the volume knob could turn a little more). Note that the improvement depends on whether the goal is to minimize or maximize the target function. For instance, in Giapetto's problem, which seeks maximization, the marginal value 1 means that the objective function would *increase* by 1 if we could have one more hour of finishing labor (we know it's one *more* hour and not one less, because the finishing hours constraint is an upper boundary).

The carpentry and soldier demand constraints are similar. For the carpentry constraint, note that it's also an upper boundary. Therefore, a relaxation of one unit in that constraint (an increment of one hour) would make the objective function's optimal value become better by the marginal value 1 and become 181.

The soldier demand, however, is not bounded, thus its state is `B`, and a relaxation in it will not change the objective function's optimal value.

Try relaxing the value of each bounded constraint one at a time, solve the modified problem, and see what happens to the optimal value of the objective function. Also check that changing the value of unbounded constraints won't make any difference to the solution, as expected.

Finally, glpsol's report shows the values for the decision variables: $x_1 = 20$ and $x_2 = 60$. This tells Giapetto that he should produce 20 soldiers and 60 trains to maximize his weekly profit.

Giapetto's problem was very small. You may be wondering, in a problem with many more decision variables and constraints, would you have to declare each variable and each constraint separately? And what if you wanted to adjust the data of the problem, such as the selling price of a soldier? Do you have to make changes everywhere this value appears? The next section discusses that.

Using model and data sections in Giapetto's problem

MathProg models normally have a model section and a data section, sometimes in two different files. Thus, glpsol can solve a model with different data sets easily, to check what the solution would be with this new data. The following listing states Giapetto's problem in a much more elegant way:

Listing 4. Giapetto's problem with a model and a data section: giapetto2.mod

```
#
# Giapetto's problem
#
# This finds the optimal solution for maximizing Giapetto's profit
#

/* Set of toys */
set TOY;

/* Parameters */
param Finishing_hours {i in TOY};
param Carpentry_hours {i in TOY};
param Demand_toys     {i in TOY};
param Profit_toys     {i in TOY};

/* Decision variables */
var x {i in TOY} >=0;

/* Objective function */
maximize z: sum{i in TOY} Profit_toys[i]*x[i];

/* Constraints */
s.t. Fin_hours : sum{i in TOY} Finishing_hours[i]*x[i] <= 100;
s.t. Carp_hours : sum{i in TOY} Carpentry_hours[i]*x[i] <= 80;
s.t. Dem {i in TOY} : x[i] <= Demand_toys[i];


data;
/* data  section */

set TOY := soldier train;

param Finishing_hours:=
soldier         2
train           1;

param Carpentry_hours:=
soldier         1
train           1;

param Demand_toys:=
soldier        40
train    6.02E+23;

param Profit_toys:=
soldier         3
```

```
47      train          2;
48
49      end;
```

Rather than two separate files, the problem is stated in a single file with a modeling section (lines 1 through 27) and a data section (lines 28 through 49).

Line 8 defines a `SET`. A `SET` is a universe of elements. For example, if I declare mathematically x_i, `for all i in {1;2;3;4}`, I'm saying that `x` is an array that ranges from 1 to 4, and therefore we have x_1, x_2, x_3, x_4. In this case, `{1;2;3;4}` is the set. So, in Giapetto's problem, there's a set called `TOY`. Where are the actual values of this set? In the data section of the file. Check line 31. It defines the `TOY` set to contain `soldier` and `train`. Wow, the set is not a numerical range. How can that be? GLPK handles this in an interesting way. You'll see how to use this in a few moments. For now, get used to the syntax for `SET` declarations in the data section:

`set` label `:=` value1 value2 ... valueN `;`

Lines 11, 12, and 13 define the parameters of the problem. There are three: `Finishing_hours`, `Carpentry_hours`, and `Demand_toys`. These parameters make up the problem's data matrix and are used to calculate the constraints, as you'll see later.

Take the `Finishing_hours` parameter as an example. It's defined on the `TOY` set, so each kind of toy in the `TOY` set will have a value for `Finishing_hours`. Remember that each soldier requires 2 hours of finishing work, and each train requires 1 hour of finishing work. Check lines 33, 34, and 35 now. There is the definition of the finishing hours for each kind of toy. Essentially, those lines declare that `Finishing_hours[soldier]=2` and that `Finishing_hours[train]=1`. `Finishing_hours` is, therefore, a matrix with 1 row and 2 columns.

Carpentry hours and demand parameters are declared similarly. Note that the demand for trains is unlimited, so a very large value is the upper bound on line 43. Does that value seem fa-*mole*-iar to you?

Line 17 declares a variable, `x`, for every `i` in `TOY` (resulting in `x[soldier]` and `x[train]`), and constrains them to be greater than or equal to zero. Once you have sets, it's pretty easy to declare variables, isn't it?

Line 20 declares the objective (target) function as the maximization of `z`, which is the total profit for every kind of toy (there are two: trains and soldiers). With soldiers, for example, the profit is the number of soldiers times the profit per soldier.

The constraints on lines 23, 24, and 25 are declared in a similar way. Take the finishing hours constraint as an example: it's the total of the finishing hours per kind of toy, times the number of that kind of toy produced, for the two types of toys (trains and soldiers), and it must be less than or equal to 100. Similarly, the total carpentry hours must be less than or equal to 80.

The demand constraint is a little bit different than the previous two, because it's defined for each kind of toy, not as a total for all toy types. Therefore, we need two of them, one for trains and one for soldiers, as you can see on line 25. Note that the index variable (`{i in TOY}`) comes before the `:`. This tells GLPK to create a constraint for each toy type in TOY, and the equation that will rule each constraint will be what comes after the `:`. In this case, GLPK will create

```
Dem[soldier] : x[soldier] <= Demand[soldier]

Dem[train] : x[train] <= Demand[train]
```

Solving this new model must yield the same results:

Listing 5. The solution to Giapetto's problem with a data section: giapetto2.sol

```
Problem:    giapetto2
Rows:       5
Columns:    2
Non-zeros:  8
Status:     OPTIMAL
Objective:  z = 180 (MAXimum)

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 z            B            180
     2 Fin_hours    NU           100                         100             1
     3 Carp_hours   NU            80                          80             1
     4 Dem[soldier] B             20                          40
     5 Dem[train]   B             60                    6.02e+23

   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 x[soldier]   B             20             0
     2 x[train]     B             60             0

Karush-Kuhn-Tucker optimality conditions:

KKT.PE: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

KKT.PB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

KKT.DE: max.abs.err. = 0.00e+00 on column 0
        max.rel.err. = 0.00e+00 on column 0
        High quality

KKT.DB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

End of output
```

Note how the constraints and the decision variables are now named after the `TOY` set, which looks clean and organized. Very good. You have maximized Giapetto's profit!

Conclusion

You've seen how to formulate a simple, two-variable linear problem. Then you saw how to use a simple **MathProg** program to solve it using sets, parameters, constraints, decision variables, and an objective (target) function. The program used summation over sets and a parameters data section. Finally, you learned how to interpret the results of a maximization problem.

The next installment in this three-article series will show you how to make the most out of a bad diet.

Download

Description	Name	Size	Download method
Solutions to the problem	solutions.zip	1KB	HTTP

Information about download methods

Resources

Learn

- The problems in this article are taken with permission from *Operations Research: Applications and Algorithms, 4th Edition*, by Wayne L. Winston (Thomson, 2004).
- The online documentation for GLPK gives more information about GLPK, how to get the software, and how to join the GLPK community.
- Check out the Wikipedia entry for GLPK.
- Subscribe to the GLPK help mailing list or bug reports mailing list.
- In the developerWorks Linux zone, find more resources for Linux developers.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- With **IBM** trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Rodrigo Ceron Ferreira de Castro is a Staff Software Engineer at the **IBM** Linux Technology Center. He graduated from the State University of Campinas (UNICAMP) in 2004. He received the State Engineering Institute prize and the Engineering Council Certification of Honor when he graduated. He's given speeches in open source conferences in Brazil and other countries.