AMTH250

Linear Equations, Linear Programming

Gary Bunting

August 22, 2011

Contents

1 Linear Equations

1.1 Theory

We will consider the problem of solving a system of m linear equations in n unknowns:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \cdots & + & a_{mn}x_n & = & b_m \end{array} \tag{1}$$

or, in matrix form,

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

where $\mathbf{A}$ is a given $m \times n$ matrix, $\mathbf{b}$ is a given m vector, and $\mathbf{x}$ is the n vector to be solved for.

The most important case is when $m = n$, that is the number of equations equals the number of unknowns. In this case the system of equations (1) has a unique solution $\mathbf{x}$ provided the matrix $\mathbf{A}$ is nonsingular, i.e. the determinant $\det \mathbf{A} \neq 0$. In this section we will assume we are dealing with a non-singular $n \times n$ system of equations.

The **rank** of a matrix $\mathbf{A}$, denoted by $\operatorname{rank} \mathbf{A}$, is the number of linearly independent rows or columns of the matrix. That the number of linearly independent rows is equal to the number of linearly independent columns is a non-trivial theorem of linear algebra. A $n \times n$ matrix $\mathbf{A}$ is non-singular if and only if $\operatorname{rank} \mathbf{A} = n$.

1.2 Solving Linear Equations

When $m = n$ the algorithm used by Octave and other software for solving linear equations is based on Gaussian elimination. Linear equations are solved in Octave using the backslash, `\`, operator. If `a` is a matrix and `b` is a vector, then

```
x = a \ b
```

is the solution of the linear system `a * x = b`.

Example

Here is a simple example with a 3×3 random system:

```
octave> a = randn(3,3)
a =
   1.5756007  -0.5321135  -1.1779000
   0.0043999   0.8795361   0.8367506
   0.0460965  -0.0068975  -0.0786655
```

```
octave:> b = randn(3,1)
b =
  -0.79410
   0.55439
  -0.44255

octave:> x = a\b
x =
   3.1547
  -7.0873
   8.0957
```

This is our solution. We can check that `x` does satisfy the equations:

```
octave:> res = a*x - b
res =
   1.1102e-16
   4.4409e-16
   0.0000e+00
```

The result is not quite zero because of rounding errors. Note that for this example these errors are of order $\varepsilon_{\text{mach}}$ but they can be much larger for large systems systems of equations.

1.3 Complexity

The the most direct measure of the amount of work involved in solving a linear system is a count of the number of arithmetic operations involved. The number of arithmetic operations required to solve an $n \times n$ system of linear equations by Gaussian elimination is approximately $\frac{2}{3}n^3$. As a consequence the time taken to solve a linear system is approximately proportional to n^3. This tells us, for instance, that solving a 100×100 system of equations will require the order of two thirds of a million arithmetic operations and will take about 1000 times as long as solving a 10×10 system.

Example

The time (in seconds) taken for a operation in Octave can be obtained by the sequence:

```
tic; operation; toc
```

We will use this to measure the time taken to solve linear systems in Octave. The sizes of the matrices used in this sort of experiment depends on the speed of your computer and the amount of memory it has. I started with a 100×100 system:

```
octave:> a = randn(100, 100);
octave:> b = randn(100, 1);
octave:> tic; x = a\b; toc
Elapsed time is 0.001809 seconds.

octave:> a = randn(200, 200);
octave:> b = randn(200, 1);
octave:> tic; x = a\b; toc
Elapsed time is 0.01277 seconds.
```

Now according to theory, if we double n the time taken to solve a linear system should increase by a factor of 8, and our results agree[1] with that. Continuing:

```
octave:> a = randn(400, 400);
octave:> b = randn(400, 1);
octave:> tic; x = a\b; toc
Elapsed time is 0.064342 seconds.

octave:> a = randn(800, 800);
octave:> b = randn(800, 1);
octave:> tic; x = a\b; toc
Elapsed time is 0.389519 seconds.

octave:> a = randn(1600, 1600);
octave:> b = randn(1600, 1);
octave:> tic; x = a\b; toc
Elapsed time is 2.7452 seconds.

octave:> a = randn(3200, 3200);
octave:> b = randn(3200, 1);
octave:> tic; x = a\b; toc
Elapsed time is 26.954 seconds.
```

This continues to agree with theory.

To get some idea of the scale of things; the number of floating point operations to solve this last system is about

```
octave:> (2/3)*(3200^3)
ans =  2.1845e+10
```

or 20 billion operations. A floating point number occupies 64 bits or 8 bytes, so the matrix in the last system occupies

[1] It won't be exactly a factor of 8. There are other factors to take into account, for example memory allocation and copying of results.

```
octave:> (3200^2)*8
ans =  81920000
```

bytes, i.e. about 80 megabytes.

1.4 Vector Norms

There are no approximations involved in Gaussian elimination, so if we could use exact arithmetic we would always get the exact solution. However we need to work with floating point arithmetic and take into account the effect of rounding errors. Given the large number of arithmetic operations involved, e.g. of the order of two thirds of a million for a 100×100 system, these cannot be ignored.

When we talk about the accuracy of the solution of a system of linear equations we want some simple measure of how close the computed solution $\hat{\mathbf{x}}$ is to the exact solution $\mathbf{x}$. One obvious measure the Euclidean distance between the two vectors:

$$||\hat{\mathbf{x}} - \mathbf{x}|| = \left[\sum_{i=1}^{n} (\hat{x}_i - x_i)^2\right]^{\frac{1}{2}}$$

More generally, for any vector $\mathbf{x}$ the **norm** (or more precisely 2-norm) of $\mathbf{x}$ is defined by

$$||\mathbf{x}|| = \left[\sum_{i=1}^{n} x_i^2\right]^{\frac{1}{2}}$$

1.5 Condition Number

When we compute the solution of a linear system we don't know the exact solution and so we have no direct way of determining the accuracy of our solution. A simple idea, which we used earlier, is to compute the **residual**

$$\mathbf{r} = \mathbf{A}\hat{\mathbf{x}} - \mathbf{b}$$

which would be zero if the computed solution where exact. Gaussian elimination tends to have the effect of computing a solution with a small residual even when the solution is wildly inaccurate, so this is at best an unreliable indicator of accuracy.

A much better indicator of accuracy is obtained from the **condition number**, $\operatorname{cond} \mathbf{A}$, of the matrix $\mathbf{A}$. This is defined by

$$\operatorname{cond} \mathbf{A} = \frac{\sigma_1}{\sigma_m}$$

the ratio of the largest to smallest *singular values*. This is a measure of close the matrix is to being singular, and it enables us to give reliable estimates

of the relative error in the computed solution:

$$\text{Rel. Error} = \frac{||\hat{\mathbf{x}} - \mathbf{x}||}{||\mathbf{x}||} \approx \text{cond}\,\mathbf{A} \times \varepsilon_{\text{mach}} \tag{2}$$

Machine epsilon appears in this expression because limited precision is the only source of error in Gaussian elimination. A matrix with large condition number is said to be **ill-conditioned**.

The approximation in Equation (2) is almost invariably an overestimate of the error, typically by a factor of about 10. This is not too bad because (a) rounding error, which what we are talking about here, is notoriously difficult to estimate, and (b) it clearly better to err on the side of caution in problems like this.

Example

Again we will use a random matrix.

```
octave:> a = randn(100, 100);
```

Now we will create a random vector `x0` and compute `b = a*x0` which will serve as the right hand side of our system of equations

```
octave:> x0 = randn(100, 1);
octave:> b = a*x0;
```

The point of this little trick is that `x0` is the exact solution of the system `Ax = b` and we can use this compute the error in our example.

First we compute the solution:

```
octave:> x = a\b;
```

The residual and its norm are:

```
octave:> res = a*x - b;
octave:> norm(res)
ans =  5.3550e-14
```

Our estimate the relative error in the computed solution is

```
octave:> cond(a)
ans =  686.83
octave:> cond(a)*eps
ans =  1.5251e-13
```

The actual relative error in this example is

```
octave:> err = norm(x - x0)/norm(x0)
err =  2.3394e-14
```

We have over-estimated the error by a factor of six. To continue with this experiment we create a script:

```
% lineqnerr: estimated and actual error for a random n x n system of
% linear equations. Note: n must be defined before script is called.
a = randn(n,n);
x0 = randn(n,1);
b = a*x0;
x = a\b;
esterr = cond(a)*eps;
err = norm(x-x0)/norm(x0);
disp("Estimated error:"), disp(esterr)
disp("Actual error:"), disp(err)
disp("Ratio estimate/actual:"), disp(esterr/err)
```

Now

```
octave:> n=10; lineqnerr
Estimated error:
 5.4605e-15
Actual error:
 7.5257e-16
Ratio estimate/actual:
 7.2559

octave:> n=100; lineqnerr
Estimated error:
 9.3074e-14
Actual error:
 1.3368e-14
Ratio estimate/actual:
 6.9622

octave:> n=1000; lineqnerr
Estimated error:
 4.9889e-13
Actual error:
 5.4030e-14
Ratio estimate/actual:
 9.2336
```

Note that (a) the size of the error tends to increase with n, but (b) the factor by which the error is over-estimated does not depend on n.

2 Eigenvalues and Eigenvectors

First we recall the definitions: if $\mathbf{A}$ be a $n \times n$ matrix, a vector $\mathbf{x}$ is an **eigenvector** of $\mathbf{A}$ with **eigenvalue** λ when

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}.$$

The Octave command `eig` is used to compute eigenvalues and eigenvectors. It can be used in two forms; the command

```
eig(a)
```

returns the eigenvalues of `a` as components of a vector. Assigning a pair of values to result:

```
[V, D] = eig(a)
```

returns the eigenvectors as *columns* of the matrix `V` and the corresponding eigenvalues as the diagonal elements of the matrix `D`.

Example

```
octave:> a = randn(3,3)
a =
  -1.059000  -1.006809  -0.482307
   1.237614  -1.159737   0.405213
  -0.017364   1.598721   0.225513

octave:> eig(a)
ans =
  -1.08636 + 0.72627i
  -1.08636 - 0.72627i
   0.17949 + 0.00000i
```

In this case the matrix has one real and two complex eigenvalues. Note that the complex eigenvalues of a real matrix come in complex conjugate pairs.

Here are the eigenvalues and eigenvectors:

```
octave:> [V,D] = eig(a)
V =

  -0.40428 - 0.28987i  -0.40428 + 0.28987i  -0.34091 + 0.00000i
  -0.52263 + 0.28376i  -0.52263 - 0.28376i  -0.03075 + 0.00000i
   0.63156 + 0.00000i   0.63156 - 0.00000i   0.93959 + 0.00000i
```

```
D =
Diagonal Matrix

  -1.08636 + 0.72627i                      0                      0
                    0  -1.08636 - 0.72627i                      0
                    0                      0   0.17949 + 0.00000i
```

We can check the answer by extracting the individual eigenvalues and eigenvectors. The first eigenvalue and corresponding eigenvector are (note the use of the colon operator to extract the first column of matrix `V`):

```
octave:> k1 = D(1,1)
k1 = -1.08636 + 0.72627i

octave:> v1 = V(:,1)
v1 =

  -0.40428 - 0.28987i
  -0.52263 + 0.28376i
   0.63156 + 0.00000i
```

Checking, using the definition of an eigenvalue and an eigenvector:

```
octave:> a*v1 - k1*v1
ans =
  1.1102e-16 + 7.9797e-17i
  1.1102e-16 + 0.0000e+00i
  -4.4409e-16 + 0.0000e+00i
```

Similar calculations could be done for the other eigenvalues and eigenvectors.

3 Linear Programming

3.1 Linear Programming Problems

Linear programming problems are a special type of optimization problem. The general linear programming problem can be formulated as follows:

Find $x_1, x_2, \ldots, x_n$ to

Minimize the **objective function**

$$c = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

Subject to the **constraints**

$$\begin{array}{ccccccc} a_{11}x_1 & + & a_{12}x_2 & +\ldots & + & a_{1n}x_n & \leq & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & +\ldots & + & a_{2n}x_n & \leq & b_2 \\ \vdots & & \vdots & & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & +\ldots & + & a_{mn}x_n & \leq & b_m \end{array}$$

The *linear* in linear programming comes from the fact that the objective function and constraints are linear functions of the variables $x_1, x_2, \ldots, x_n$.

Instead of the inequality $\leq$ in the constraints, the inequality $\geq$ or equality $=$ may be used. This seemingly greater generality does not lead to a larger class of problems since a constraint of the form $a \geq b$ can be converted to $-a \leq -b$ and an equality constraint $a = b$ is equivalent to the pair of constraints $a \leq b$ and $-a \leq -b$. Similarly maximizing an object function c is equivalent to minimizing the function $-c$.

The **feasible region** is the set of points $(x_1, \ldots, x_n) \in \mathbb{R}^n$ satisfying the constraints of the problem.

3.2 An Example

A transport company needs to move sand between three sources X, Y and Z and three destinations A, B and C. Each source has a limited supply of sand:

Source	Truckloads available
X	35
Y	40
Z	40

Each destination has a minimum requirement:

Destination	Truckloads required
A	45
B	50
C	15

Transportation costs, in $ per truckload, are:

	A	B	C
X	5	10	10
Y	20	30	20
Z	5	8	12

The problem is to find a transportation schedule to minimize the cost of transporting the sand.

The formulation of a linear programming problem requires three steps:

1. Identify the **variables**.
2. Write down the **objective function**.
3. Write down the **constraints**.

In this problem there are nine variables; the number of truckloads of sand moved between each of the three sources and each of the three destinations. Denote these variables by N_{XA}, N_{XB}, N_{XC}, N_{YA}, N_{YB}, N_{YC}, N_{ZA}, N_{ZB} and N_{ZC}, where, for example, N_{YA} denotes the number of truckloads of sand transported from Y to A.

The objective is to minimize the total transportation cost. Since we know the cost of transporting one truckload of sand from any source to any destination, it is easy to express the cost as a function of the variables:

$$\begin{aligned} c = & 5N_{XA} + 10N_{XB} + 10N_{XC} + 20N_{YA} + 30N_{YB} + 20N_{YC} + \\ & 5N_{ZA} + 8N_{ZB} + 12N_{ZC} \end{aligned}$$

There are two sets of constraints relating, respectively, to the supply and demand for sand.

Constraints on supply:

$$\begin{aligned} N_{XA} + N_{XB} + N_{XC} &\leq 35 \\ N_{YA} + N_{YB} + N_{YC} &\leq 40 \\ N_{ZA} + N_{ZB} + N_{ZC} &\leq 40 \end{aligned}$$

Constraints on demand:

$$\begin{aligned} N_{XA} + N_{YA} + N_{ZA} &= 45 \\ N_{XB} + N_{YB} + N_{ZB} &= 50 \\ N_{XC} + N_{YC} + N_{ZC} &= 15 \end{aligned}$$

Finally, there are the easily overlooked constraints that each of the variables is non-negative:

$$\begin{array}{lll} N_{XA} \geq 0 & N_{XB} \geq 0 & N_{XC} \geq 0 \\ N_{YA} \geq 0 & N_{YB} \geq 0 & N_{YC} \geq 0 \\ N_{ZA} \geq 0 & N_{ZB} \geq 0 & N_{ZC} \geq 0 \end{array}$$

Requiring all variables to be non-negative is very common in linear programming problems.

3.3 Geometry of 2D Problems

We will look at a two dimensional linear programming problem which illustrates the geometric side to the problem:

Minimize

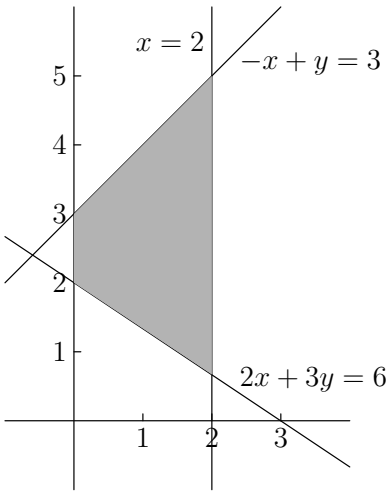
$$c = -2x + y$$

Subject to

$$\begin{aligned} 2x + 3y &\geq 6 \\ -x + y &\leq 3 \\ x &\leq 2 \\ x \geq 0 \quad y &\geq 0 \end{aligned}$$

The feasible region for this problem is shown Figure 1.

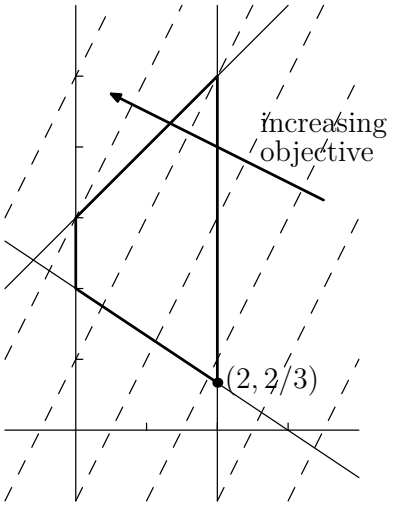
Figure 1: Feasible region



Recall that the level sets of a function $f(x_1, x_2, \ldots, x_n)$ are the sets $f(x_1, x_2, \ldots, x_n) =$ constant. The level sets of the objective function $c = -2x + y$ are the lines

$$-2x + y = \text{ constant}$$

Figure 2: Level Sets of the Objective Function



From this diagram it is apparent that the minimum of the objective function occurs at the vertex $(2, 2/3)$ of the feasible region.

This example is typical of linear programming problems in two variables:

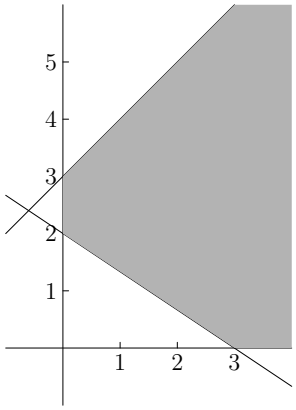
1. The feasible region is a convex polygonal region of the plane bounded by the constraints.

2. If the feasible region is non-empty and bounded, then the linear programming problem has a solution.

3. The solution, if it exists, occurs at a vertex of the feasible region.

If we remove the constraint $x \leq 2$ we obtain an unbounded feasible region in which our objective function has no minimum. See Figure 3.3.

Figure 3: An Unbounded Feasible Region



3.4 The Simplex Algorithm

Geometrically, the general linear programming problem is similar to the two dimensional problem.

1. The feasible region is a convex polygonal region of $\mathbb{R}^n$. It is bounded by hyperplanes, i.e. $(n-1)$-dimensional planes in $\mathbb{R}^n$, determined by the constraints.
2. The level sets of the objective function are also hyperplanes in $\mathbb{R}^n$.
3. If the feasible region is non-empty and bounded, then the linear programming problem has a solution.
4. The solution, if it exists, occurs at a vertex of the feasible region.

The standard method for solving linear programming problems is the **Simplex Algorithm**. The algorithm is easy to describe: it simply searches through the vertices of the feasible region in a systematic way until the minimum is found. Practical linear programming problems with thousands of variables and constraints are not unusual.

3.5 Integer Programming

When it is required that all the variables in a LP problem be integers then the problem is called an **integer programming** or IP problem. If only some of the variables are required to be integers then it is called a **mixed integer programming** problem. In contrast to linear programming where there is an efficient algorithm — the simplex algorithm — for solving large problems, large integer and mixed integer programming problems are computationally difficult.

Example

Let us return to the problem of §3.3, this time with the requirement that the variables be integers.

Minimize

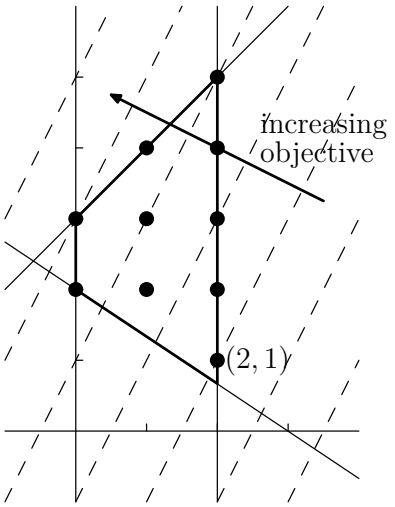
$$c = -2x + y$$

Subject to

$$\begin{aligned} 2x + 3y &\geq 6 \\ -x + y &\leq 3 \\ x &\leq 2 \\ x \geq 0 \quad y &\geq 0 \end{aligned}$$

Figure 3.5 shows the geometry of the problem with integer points in the feasible region marked with dots. From the figure we see that the optimal solution occurs at $x = 2, y = 1$.

Figure 4: Feasible Points for an Integer Programming Problem



3.6 Octave

Octave's linear programming solver is `glpk`. It is used in the form

```
[x, opt] = glpk(obj, cnstr, rhs, lb, ub, ctype, vtype, ptype)
```

We will illustrate its use on the two dimensional problem of §3.3:

1. The first argument, `obj`, is a *column* vector containing the coefficients of the objective function.

```
octave:> obj = [-2 1]'
obj =
  -2
   1
```

2. The second argument, `cnstr`, is a matrix containing the coefficients of the constraints. Each row of the matrix contains the coefficients of one constraint.

```
octave:> cnstr = [ 2 3; -1 1; 1 0]
cnstr =
   2   3
  -1   1
   1   0
```

3. The argument `rhs` is the *column* vector containing the coefficients on the right hand side of the constraints.

```
octave:> rhs = [6 3 2]'
rhs =
   6
   3
   2
```

4. The two arguments, `lb` and `ub` are column vectors of lower bounds and upper bounds on the variables. In our example both variables have lower bounds of zero and no upper bounds (other than those determined by the constraints). This a common situation in linear programming. If there are no bounds of some type an empty matrix, `[]` can be used.

```
octave:> lb = [0 0]'
lb =
   0
   0
octave:> ub = []
ub = []
```

5. The argument `ctype` determines the type of each constraint. The code for the constraints are:

   - S for a $=$ constraint.
   - U for a $\leq$ constraint.
   - L for a $\geq$ constraint.

These codes are combined into a string, for our example:

```
octave:> ctype = "LUU"
ctype = LUU
```

6. The argument `vtype` determines the type of each variable, either C for a continuous (real) variable or I for an integer variable. We will solve the problem as both a LP and an IP problem:

```
octave:> vtype1 = "CC", vtype2 = "II"
vtype1 = CC
vtype2 = II
```

7. The argument `ptype` takes the value 1 for a minimization problem or -1 for a maximization problem.

```
octave:> ptype = 1
ptype =  1
```

8. Finally, the return values `[x,opt]` give the optimal solution x and the value of the objective function at the optimum.

Now we can compute the solution:

```
octave:> [x, opt] = glpk(obj, cnstr, rhs, lb, ub, ctype, vtype1, ptype)
x =
   2.00000
   0.66667
opt = -3.3333

octave:> [x, opt] = glpk(obj, cnstr, rhs, lb, ub, ctype, vtype2, ptype)
x =
   2
   1
opt = -3
```

which agree with the solutions we found earlier using geometric reasoning.

Example

We will now use Octave solve the problem of §3.2

In our problem there are nine variables: N_{XA}, N_{XB}, N_{XC}, N_{YA}, N_{YB}, N_{YC}, N_{ZA}, N_{ZB} and N_{ZC}. We will take the variables in this order so the objective is:

```
octave:> obj = [5 10 10 20 30 20 5 8 12]';
octave:> vtype = "CCCCCCCCC";
octave:> ptype = 1;
```

There are six constraints. The variables appearing in each constraint follow a pattern:

```
octave:> cnstr = [1 1 1 0 0 0 0 0 0
> 0 0 0 1 1 1 0 0 0
> 0 0 0 0 0 0 1 1 1
> 1 0 0 1 0 0 1 0 0
> 0 1 0 0 1 0 0 1 0
> 0 0 1 0 0 1 0 0 1]
octave:> rhs = [35 40 40 45 50 15]';
octave:> ctype = "UUUSSS";
```

Each variable has a lower bound of zero, and there are no explicit upper bounds:

```
octave:> lb = [0 0 0 0 0 0 0 0 0]';
octave:> ub =[];
```

Now we can solve the problem in octave:

```
octave:25> [x, opt] = glpk(obj, cnstr, rhs, lb, ub, ctype, vtype, ptype)
x =
   25
   10
    0
   20
    0
   15
    0
   40
    0
opt =  1245
```

This result can be summarized in the table:

	A	B	C	Total
X	25	10	0	35
Y	20	0	15	35
Z	0	40	0	40
Total	45	50	15	

with optimal cost $1245.