Optimizing with Defined Operators

By Stephen M. Mansour

Abstract

Mathematical programming can be used to optimize. The typical mathematical notation for optimization is: $\max_{x} c'x$ subject to $Ax \leq b$ for linear programming (LP) [1] or $\min_{x} f(x)$ subject to $g(x) \geq 0$ for non-linear programming (NLP). We can create similar expressions using standard APL syntax.

We propose the following syntax for linear programming (LP) :

```
NS← ⌊ optimize C x subjectTo A x ≥ B   ⍝ Expression 1

NS ← ⌈ optimize C x subjectTo A x ≤ B   ⍝ Expression 2
```

We propose the following syntax for non-linear programming (NLP)

```
NS ←  ⌊ optimize f x subjectTo g x ≥ 0     ⍝ Expression 3

NS ←  ⌈ optimize f x subjectTo g x ≤ 0     ⍝ Expression 4
```

Parsing Expressions 2 and 3 above, we arrive at the following:

```
   NS ←   (⌈ optimize) (C x subjectTo) (A x ≤)     B      ⍝ LP

   NS ←   (⌊ optimize) (f x subjectTo) (g x ≥)     0      ⍝ NLP
 --------  -----------  --------------  ------- ---------  -----
    ↑         ↑              ↑              ↑         ↑
  Result     Runs the      builds a     creates a   right
 Namespace   LP/NLP        tableau      namespace   arg
```

We need to apply some sleight-of-hand to make the syntax work. Since `x` represents a vector of decision variables, it is unknown at the beginning of the optimization process. So, we don't need to assign any values to it. As the middle item in a function expression, `x` must be either a function in a 3-train (fork), or a dyadic operator. If `x` is the middle item in a fork, we would be required to keep the parentheses; and parsing would be difficult. If we make `x` a dyadic operator, binding rules eliminate the parenthesis and preserve the arguments. Let's look first at the syntax of the rightmost function expression:

```
   NS          ←  ( A        x         ≤    )      b        ⍝ LP
   NS          ←  ( G        x         ≥    )      0        ⍝ NLP
   ↑              ↑         ↑         ↑           ↑
  Result         Left    operator right        right
 Namespace      operand           operand      argument
```

[1] In mathematics a vector is represented as an nx1 matrix. The notation c' in mathematics mean the transpose of the column vector c which results in a 1xn row vector c' . In APL, this is not necessary because inner product handles vectors and matrices in a more natural way.

Note that the left operand is the array A in the LP case, and the function-array G in the NLP case. The function derived from this takes a vector right argument b representing the right-hand-sides of the constraint inequalities, or the scalar 0 in the NLP case. The result of this derived function is a namespace containing the following items:

LP: Linear Program	NLP: Non-Linear Program
NS.A: matrix of constraint coefficients	NS.G function array
NS.b: vector of right hand sides	NS.b 0
NS.rel: relation function	NS.rel: relation function

The middle function expression takes this namespace NS as a right argument, and builds a tableau from the feasible region defined by the variables A and b and the function rel in the namespace. (For NLP, the feasible region is defined by the function G.)

```
       NS ← ( c        x       subjectTo )    NS           ⍝ LP
       NS ← ( f        x       subjectTo )    NS           ⍝ NLP
  -------    --     --------  ----------     ---------
       ↑      ↑        ↑           ↑          ↑
 Updated     Left   operator     right      Parameter
 Namespace  operand             operand    Namespace
```

Notice we are using the same operator x as in the rightmost function expression. But this function expression takes a namespace as its right argument, whereas previously the right argument was a simple numeric vector (or scalar). The operator x can check the name class of its right argument to determine how to proceed.

```
        NS ←  ( ⌈      optimize )   NS
        NS ←  ( ⌊      optimize )   NS
       ---    ----     --------    ---
        ↑       ↑         ↑         ↑
    Solution   Left     operator   right
    Namespace  operand             argument
```

We now apply the operator optimize function to the namespace created by applying the x operator twice. The left operand ⌊ or ⌈ determines whether to minimize or maximize the objective. The result is the updated operator which now contains the following variables:

```
NS.Decision     ⍝ Optimal value of Decision Variables (Vector)
NS.Objective    ⍝ Value of objective function (Scalar)
NS.ShadowPrice  ⍝ Increase/Decrease in objective function (vector)
NS.ReducedCost  ⍝ Profit contribution minus resource use (vector)
```

Example 1: Blue Ridge Hot Tubs

A manufacturer produces three types of hot tubs:

Hot Tub Brand:	Aqua-Spa	Hydro-Luxe	Typhoon-Lagoon	Resources Available
Unit Profit:	\$350	\$300	\$320	
Pumps Required	1	1	1	200
Labor Required	9 hours	6 hours	8 hours	1566
Tubing Needed	12 feet	16 feet	13 feet	2880

We formulate the problem as follows:

$X_1 =$ Number of Aqua-Spas to produce
$X_2 =$ Number of Hydro-Luxes to produce
$X_3 =$ Number of Typhoon-Lagoons to produce

$$\begin{aligned}
&\text{Maximize } 350X_1 + 300X_2 + 320X_3 \\
&\text{Subject to: } \quad X_1 + X_2 + X_3 \le 200 \\
&\qquad 9X_1 + 6X_2 + 8X_3 \le 1{,}566 \\
&\qquad 12X_1 + 16X_2 + 13X_3 \le 2{,}880 \\
&\qquad X_1, X_2, X_3 \ge 0
\end{aligned}$$

Using matrix notation, we can define the problem mathematically as follows:

$$x = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} \quad c = \begin{bmatrix} 350 \\ 300 \\ 320 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 1 & 1 \\ 9 & 6 & 8 \\ 12 & 16 & 13 \end{bmatrix} \quad b = \begin{bmatrix} 200 \\ 1566 \\ 2880 \end{bmatrix}$$

Maximize $c'x$ subject to $Ax \le b$

We can now do the same thing in APL and obtain a solution:

```
      C←350 300 320                         ⍝ Objective coefficients
      ⎕← A←3 3⍴1 1 1 9 6 8 12 16 13         ⍝ Constraint coefficients
 1  1  1
 9  6  8
12 16 13
      B←200 1566 2880                       ⍝ Resource limitations
      NS←⌈ optimize C x subjectTo A x ≤ B   ⍝ Perform the LP
      NS.Decision     ⍝ Produce 122 Aqua Spas and 78 Hydro-Luxes
122 78 0
      NS.Objective    ⍝ Total profit $66,100
66100
      NS.ShadowPrice  ⍝ Each add’l pump contributes $200 to profit
200 16.66666667 0     ⍝ Each add’l labor hour contributes $16.67 profit
      NS.ReducedCost  ⍝ Each Typhoon-Lagoon produced reduces profit by $13.33
0 0 ¯13.33333333
```

Example 2: Weedwacker Company – Make or Buy

The company produces two types of law trimmers; an electric and a gas model. The table below indicates the requirements and production capability:

	Electric Trimmers	Gas Trimmers	Total Hours Available
Production	0.20 hours	0.40 hours	10,000
Assembly	0.30 hours	0.50 hours	15,000
Packaging	0.10 hours	0.10 hours	5,000
Cost to Make	$55	$85	
Cost to Buy	$67	$95	

Number required	15,000	30,000	

We formulate this problem as follows:

M_1= number of electric trimmers to make M_2= number of gas trimmers to make
B_1= number of electric trimmers to buy B_2= number of gas trimmers to buy

Minimize $55M_1 + 85\,M_2 + 67\,B_1 + 95\,B_2$

ST $M_1 + B_1 = 30{,}000$

$M_2 + B_2 = 15{,}000$

$0.20M_1 + 0.40M_2 \le 10{,}000$

$0.30M_1 + 0.50M_2 \le 15{,}000$

$0.10M_1 + 0.10M_2 \le 5{,}000$

$M_i, B_i \ge 0$

We solve this problem in Dyalog APL as follows:

```
     C←55 85 67 95                  ⍝ Objective coefficients
     ⎕←A←5 4⍴1 0 1 0 0 1 0 1 .2 .4 0 0 .3 .5 0 0 .1 .1 0 0
1   0   1 0
0   1   0 1
0.2 0.4 0 0
0.3 0.5 0 0
0.1 0.1 0 0
     B←30000 15000 10000 15000 5000 ⍝ Resource constraints
     rel←=,=,≤,≤,≤                  ⍝ Relations (function-train)
     NS←minimize C x subjectTo A x rel B
     NS.Decision
30000 10000 0 5000  ⍝ Make 30K electric and 10K gas trimmers; buy 5K gas
     NS.Objective   ⍝ Total cost $2,975,000
2975000
     NS.ShadowPrice ⍝ Each addt'l production hour reduces cost by $25.00
60 95 ¯25 0 0
     NS.ReducedCost ⍝ Increased cost to buy one more Electric Trimmer $7.00.
0 0 7 0
```

Cover Functions

For mathematical programming purists, one may want to define the functions maximize and minimize as follows:

```
maximize ← ⌈ optimize
minimize ← ⌊ optimize
```

That way one could enter the following APL expression which mirrors the standard mathematical expression:

```
NS ← maximize c x subjectTo A x ≤ b
```

The functions `lp`, `ip`, `tp` and `nlp` were designed to encapsulate the objectives and constraints in a namespace and update it with the values of the decision variables as well as other items such as shadow prices and reduced costs. The syntax is very simple:

```
NS ← lp NS          ⍝ Linear program
NS ← ip NS          ⍝ Integer program
NS ← tp NS          ⍝ Transportation problem
```

Example 3: Garden City Beach – How Many Lifeguards?

Each summer, the city hires lifeguards to assign five consecutive days each week followed by two days off. The city's insurance company requires the minimum number of lifeguards each day:

Day	Sunday Day 0	Monday Day 1	Tuesday Day 2	Wednesday Day 3	Thursday Day 4	Friday Day 5	Saturday Day 6
Lifeguards Required	18	17	16	16	16	14	19

The city would like to determine the minimum number of lifeguards that will have to be hired. Let

Let $X_i =$ Number of workers who start on the following Day: i.e. Day `7|i+1`

For example $X_1 =$ Number of workers who start on Tuesday (Day 2)

We formulate the problem thus:

MIN $X_0 + X_1 + X_2 + X_3 + X_4 + X_5 + X_6$

ST
$X_1 + X_2 + X_3 + X_4 + X_5 \geq 18$
$X_2 + X_3 + X_4 + X_5 + X_6 \geq 17$
$X_0 + X_3 + X_4 + X_5 + X_6 \geq 16$
$X_0 + X_1 + X_4 + X_5 + X_6 \geq 16$
$X_0 + X_1 + X_2 + X_5 + X_6 \geq 16$
$X_0 + X_1 + X_2 + X_3 + X_6 \geq 14$
$X_0 + X_1 + X_2 + X_3 + X_4 \geq 19$
$X_i \geq 0$

We can create a namespace to contain all the components:

```
      EX3←⎕ns ''                    ⍝ Create namespace
      ⎕←EX3.A←(-⍳7)⌽⍤0 1⊢1 5 1/0 1 0
0 1 1 1 1 1 0
0 0 1 1 1 1 1
1 0 0 1 1 1 1
1 1 0 0 1 1 1
1 1 1 0 0 1 1
1 1 1 1 0 0 1
1 1 1 1 1 0 0

      EX3.B←18 17 16 16 16 14 19  ⍝ Constraint right hand side
```

```
      ⎕←EX3.C←7/1                  ⍝ Objective coefficients
1 1 1 1 1 1 1

      EX3.optimum←⌊                ⍝ Objective is minimum required lifeguards

      EX3.rel←≥,≥,≥,≥,≥,≥,≥        ⍝ Constraints are greater than or equal

      EX3←LP EX3                   ⍝ Perform the linear optimization

     EX3.Decision                  ⍝ Lifeguards required from each “shift”
4.6 1.6 5.6 1.6 5.6 3.6 0.6

     EX3.Objective                 ⍝ Minimum required lifeguards
23.2
```

The problem is that we get a fractional solution, whereas we only hire full-time lifeguards. We could round up the number of workers but that would not be optimal:

```
     ⌈EX3.Decision                 ⍝ Round up each shift
5 2 6 2 6 4 1
     +/⌈EX3.Decision               ⍝ Total number of lifeguards

26
```

We really want an integer solution. To accomplish this, we use integer programming by including the following constraint:

$X_i \geq 0$ & integer

```
      EX3←ip EX3                   ⍝ Run integer program
      EX3.Decision                 ⍝ Lifeguards required for each shift
3 3 5 0 8 2 3
      EX3.Objective                ⍝ Total number of lifeguards needed.
24
```

Example 4: Transportation Problem – Bonner Electronics

Bonner Electronics is planning to ship product from its Manufacturing plants in Minneapolis, Pittsburgh and Tucson to four warehouses in Atlanta, Boston, Chicago and Denver. The following table shows the unit shipping cost between each plant and warehouse:

	Warehouse				
Plant	Atlanta	Boston	Chicago	Denver	Supply
Minneapolis	$0.60	$0.56	$0.22	$.40	9.000
Pittsburgh	0.36	0.30	0.28	0.58	12.000
Tucson	0.65	0.68	0.55	0.42	13,000
Demand	7,500	8,500	9.500	8,000	

How many units must be shipped from each plant to each warehouse? There are three Supply nodes and 4 demand nodes; each of these represents a constraint. The decision variables are represented by arcs connecting each supply node to each demand node. We could set this up as a traditional LP, but it is easier to treat this as a specialized network problem known as the transportation problem.

```
     Supply←9000 12000 13000
     Demand←7500 8500 9500 8000

     ⎕←UnitCost←3 4⍴0.6 0.56 0.22 0.4 0.36 0.3 .28 .58 0.65 0.68 0.55 0.42
0.6  0.56 0.22 0.4
0.36 0.3  0.28 0.58
0.65 0.68 0.55 0.42


     (UnitCost,Supply)⍪Demand,0            ⍝ Assemble matrix
   0.6     0.56    0.22    0.4   9000
   0.36    0.3     0.28    0.58 12000
   0.65    0.68    0.55    0.42 13000
7500    8500    9500    8000        0

   EX4←TP (UnitCost,Supply)⍪Demand,0      ⍝ Transportation Problem
   EX4.Decision      ⍝ Solution e.g. ship 4000 units from Tucson to Atlanta
   0    0 9000    0
3500 8500    0    0
4000    0  500 8000


      EX4.Objective                        ⍝ Minimum total cost $12,025
12025
```

Conclusion

Various types of linear programming problems can be solved using APL operators. The functions are located in a workspace called ALPS (A Linear Programming System). References for each example are listed below. The non-linear portion is not yet available as it is still under design and development.

References

[Example 1] Ragsdale, *Spreadsheet Modeling & Decision Analysis, 7th Ed*. Cengage, 2015p. 150

[Example 2] Ibid, Chapter 3, Problem 22, p. 119

[Example 3] Ibid, Chapter 6, Problem 8, p. 292

[Example 4] *Powell, Baker, Management Science, 3rd Ed*., Wiley, 2009, P. 282