Paper 288

Advances in Mathematical Programming and Optimization in the SAS System

Trevor D. Kearney
SAS Institute Inc.
Cary, North Carolina, USA

Abstract

Optimization continues to grow in importance as the tools become more reliable and handle larger, more difficult problems. As a result, new applications with more variables and more complicated models are being introduced. The SAS® System has numerous optimization routines which handle the standard problems such as linear and nonlinear programming, integer programming, network flows, linear and nonlinear regression with all types of constraints, as well as problems with highly specialized structure, such as linear programs with embedded networks. These capabilities are exposed to the users in a variety of places such as in the procedures LP, NLP, NLMIXED, and IML, and in applications like neural networks and regression in Enterprise Miner software. This presentation gives an overview of the optimization capabilities in the SAS System, how they are changing with the availability of important new techniques, and where they will be in future releases.

Introduction

Many of the products that constitute the SAS System have procedures that employ optimization routines. For example, in SAS/STAT® , there are various procedures that use optimization to do regression. In the first section of this paper, the procedures of the SAS/OR® product are described. SAS/OR software allows optimization problems to be submitted for solution without regard to the application (statistics, forecasting, logistics, distribution, production, etc.) the model formulates.

The remaining sections of this paper describe features that have been added to the NETFLOW and LP procedures for the Version 7 release of SAS/OR software.

PROC NETFLOW now has the Interior Point (IntPoint) algorithm, an extremely efficient algorithm to solve linear programs. LP problems with over a million variables and several hundred thousand constraints are routinely solved in an hour or two on a personal computer; an impossible task for methods previously used. Using PROC NETFLOW's IntPoint algorithm is demonstrated in the second section of this paper.

PROC LP has also been the recipient of considerable development. In particular, it's integer programming facilities have had many improvements. Some solution time reductions are highlighted in the third section of this paper.

There have been improvements to the NLP procedure and new staff have been employed to continue that work and enable new or better optimizers to be available to procedures outside SAS/OR software. That could be the topic of a future paper.

Optimization in SAS/OR Software

SAS/OR software can be used to solve a wide variety of optimization problems. The basic optimization problem is that of minimizing or maximizing an objective function subject to constraints imposed on the variables of that function. The objective function and constraints can be linear or nonlinear; the constraints can be bound constraints, equality or inequality constraints, or integer constraints.

Traditionally, optimization problems are divided into Linear Programming (LP; all functions are linear) and Nonlinear Programming (NLP). Variations of LP problems are assignment problems, network flow problems, and transportation problems. Nonlinear Regression (fitting a nonlinear model to a set of data and the subsequent statistical analysis of the results) is a special NLP problem. Since these applications are so common, SAS/OR software has separate procedures or facilities within procedures for solving each type of these problems. Model data are supplied in a form suited for the particular type of problem. Another benefit is that an optimization algorithm can be specialized for the particular type of problem, reduc-

ing solution times. Optimizers can exploit some structure in problems such as imbedded networks, special ordered sets, least squares, and quadratic objective functions.

SAS/OR software has five procedures used for optimization:

- **PROC ASSIGN** solves assignment problems
- **PROC LP** solves linear and mixed integer programming problems
- **PROC NETFLOW** solves network programming problems with side constraints, and LP problems
- **PROC NLP** solves non-linear programming problems
- **PROC TRANS** solves transportation problems

PROC LP

The LP procedure solves linear and mixed integer programs. It can perform several types of post-optimality analysis, including range analysis, sensitivity analysis, and parametric programming. The procedure can also be used interactively.

PROC LP requires a problem data set that contains the model. In addition, a primal and active data set can be used for warm starting a problem that has been partially solved previously.

The problem data describing the model can be in one of two formats: a sparse or a dense format. The dense format represents the model as a rectangular matrix. The sparse format represents only the nonzero elements of a rectangular matrix.

A small product mix problem serves as a starting point for a discussion of different types of model formats supported in SAS/OR software. A candy manufacturer makes two products: chocolates and toffee. There are several constraints imposed by limitations of the manufacturing plant.

Dense format

Optimal values for two decision variables CHOCO and TOFFEE must be found. These values are the amounts of chocolate and toffee that should be made. The variable _id_ names the rows in the problem data set. The variable _type_ contains keywords that describes the type of each row in the problem data set. And the variable _rhs_ contains the right-hand-side values.

```
data dense;
   input _id_ $ CHOCO TOFFEE _type_ $ _rhs_;
   datalines;
objectiv    0.25    0.75   MAX     .
process1   15.00   40.00   LE  27000
process2    0.00   56.25   LE  27000
process3   18.75    0.00   LE  27000
process4   12.00   50.00   LE  27000
;
```

Envoke the LP procedure by specifying

```
proc lp
   data=dense;
run;
```

Sparse Format

Typically, mathematical programming models are sparse. That is, few of the coefficients in the constraint matrix are nonzero. The dense problem format shown in the last section would be an inefficient way to represent sparse models. The LP procedure also accepts data in a sparse input format. Only the nonzero coefficients need be specified.

Although the factory example of the last section is not sparse, illustrated here is an example of the sparse input format. The sparse data set has four variables: a row type identifying variable, a row name variable, a column name variable, and a coefficient variable.

```
data sparse;
   format _type_ $8. _row_ $16. _col_ $16.;
   input _type_ $_row_ $ _col_ $ _coef_ ;
   datalines;
max      object     .            .
.        object     chocolate    .25
.        object     toffee       .75
le       process1   .            .
.        process1   chocolate    15
.        process1   toffee       40
.        process1   _RHS_        27000
le       process2   .            .
.        process2   toffee       56.25
.        process2   _RHS_        27000
le       process3   .            .
.        process3   chocolate    18.75
.        process3   _RHS_        27000
le       process4   .            .
.        process4   chocolate    12
.        process4   toffee       50
.        process4   _RHS_        27000
;
```

Notice that the _type_ variable contains keywords as for the dense format, the _row_ variable contains the row names in the model, the _col_ variable contains the column names in the model, and the _coef_ variable contains the coefficients for that particular row and column.

The SPARSEDATA option in the PROC LP statement tells the LP procedure that the model in the problem data set is in the sparse format. This example also illustrates how the solution of the linear program is saved in an output data sets, the primal data set.

```
proc lp
   data=sparse sparsedata
   primalout=primal;
run;
proc print;
   var _var_ _price_ _type_ _value_ _r_cost_;
run;
```

The primal data set (Figure 1) contains the solution.

The SAS System

Obs	_VAR_	_PRICE_	_TYPE_	_VALUE_	_R_COST_
1	chocolate	0.25	NON-NEG	1000	-0.000000
2	toffee	0.75	NON-NEG	300	0.000000
3	process1	0.00	SLACK	0	-0.012963
4	process2	0.00	SLACK	10125	0.000000
5	process3	0.00	SLACK	8250	0.000000
6	process4	0.00	SLACK	0	-0.004630
7	PHASE_1_OBJECTIV	0.00	OBJECT	0	0.000000
8	object	0.00	OBJECT	475	0.000000

Figure 1. Primal data set

PROC NETFLOW

Network models describe a wide variety of real-world applications ranging from production, inventory, and distribution problems to financial applications. These models are conceptionally easy since they are based on network diagrams that represent the problem pictorially. This procedure accepts the network specification in a format that is particularly suited to networks. This not only simplifies problem description but also aids in the interpretation of the solution.

The network is represented in two data sets: a node data set that names the nodes in the network and gives supply and demand information at them, and an arc data set that defines the arcs in the network using the node names and gives arc costs and capacities.

The task PROC NETFLOW performs is to find a flow pattern through the network that is both feasible and optimal. In addition to obeying the network *flow conservation constraints*, you can specify additional *side constraints* that must also be obeyed. You provide to PROC NETFLOW the data for side constraints in an additional input data set. This data set can be specified in either the sparse or dense input formats; formats that resemble the sparse or dense input formats used by PROC LP. The model building techniques that apply to models for PROC LP also apply for network flow models having side constraints.

The side constraints can have variables directly related to the network (called *arc variables*) as well as variables (called *nonarc variables*) that have nothing to do with the network.

Flow conservation constraints should not be specified as side constraints. PROC NETFLOW was originally designed to efficiently solve LPs that have many flow conservation constraints that, if removed, are replaced by a network and a small number of side constraints. Even if your problem does not have that characteristic, PROC NETFLOW automatically detects there is no network component and solves the LP problem using the IntPoint algorithm. PROC NETFLOW's ability to solve LP problems is an important ability new for the Version 7 release of the SAS System.

But, back to the case when there is a network in the model; because the IntPoint algorithm has been implemented in PROC NETFLOW, you can choose that the IntPoint algorithm is to do the optimization, rather than the Simplex algorithm. The IntPoint algorithm is often faster when problems have many side constraints. Indicate that the IntPoint algorithm is to be used by specifying the INTPOINT option in the PROC NETFLOW statement. The input data is the same whether the Simplex or IntPoint method is used. In other aspects of how PROC NETFLOW is used, there are only minor changes. The output data set where the solution is sent is similar no matter which optimizer is used.

Networks and the Simplex Network Algorithm

Although network problems could be solved by PROC LP, the NETFLOW procedure generally solves network flow problems more efficiently than PROC LP. It does this by exploiting algebraic features of networks. Standard LP Simplex algorithms use in computations a large basis matrix. PROC NETFLOW's Simplex algorithm has been specialized for networks so that a spanning tree data structure replaces much, if not all, tha basis matrix. Computations are executed much faster, solution times are reduced.

Network Format

Consider the simple transshipment problem as an illustration.

Suppose the candy manufacturing company has two factories, two warehouses, and three customers for chocolate. The two factories each have a production capacity of 500 pounds per day. The three customers have demands of 100, 200, and 50 pounds per day, respectively.

The following data set describes the supplies (positive values for the supdem variable) and the demands (negative values for the supdem variable) for each of

the customers and factories.

```
data nodes;
   input _node_&$11. _supdem_;
   datalines;
customer_1   -100
customer_2   -200
customer_3    -50
factory_1     500
factory_2     500
;
```

Suppose there are two warehouses that are used to store the chocolate before shipment to the customers and that there are different costs for shipping between each factory, warehouse, and customer.

What is the minimum cost routing for supplying the customers?

Although not required in this example, if there were lower and upper bounds on the flow that can traverse each arc, these can be specified in the following data set. Also, a name could be given to each arc to associate it with constraint data.

```
data network;
   input _from_&$11. _to_&$11. _cost_ ;
   datalines;
factory_1    warehouse_1  10
factory_2    warehouse_1   5
factory_1    warehouse_2   7
factory_2    warehouse_2   9
warehouse_1  customer_1    3
warehouse_1  customer_2    4
warehouse_1  customer_3    4
warehouse_2  customer_1    5
warehouse_2  customer_2    5
warehouse_2  customer_3    6
;
```

To envoke PROC NETFLOW, have it save the solution (Figure 2) in the arc_sav data set, use

```
proc netflow
   arcdata=network nodedata=nodes
   arcout=arc_sav;
run;
proc print;
   var _from_ _to_ _cost_ _supply_
       _demand_ _flow_ _fcost_ _rcost_;
   sum _fcost_;
run;
```

The optimal flows that should be conveyed by each arc are values of the _FLOW_ variable. _FCOST_ values are the multiple of _FLOW_ and the arc's cost. If all these _FCOST_ values are summed, the result is the total solution cost.

Obs	_from_	_to_	_cost_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_	_RCOST_
1	warehouse_1	customer_1	3	.	100	100	300	.
2	warehouse_2	customer_1	5	.	100	0	0	4
3	warehouse_1	customer_2	4	.	200	200	800	.
4	warehouse_2	customer_2	5	.	200	0	0	3
5	warehouse_1	customer_3	4	.	50	50	200	.
6	warehouse_2	customer_3	6	.	50	0	0	4
7	factory_1	warehouse_1	10	500	.	0	0	5
8	factory_2	warehouse_1	5	500	.	350	1750	.
9	factory_1	warehouse_2	7	500	.	0	0	.
10	factory_2	warehouse_2	9	500	.	0	0	2
							=======	
							3050	

Figure 2. ARCOUT data set

Notice which arcs have positive flow (_FLOW_ is greater than 0). These arcs indicate the amount of chocolate that should be sent from factory_2 to warehouse_1 and from there on to the three customers. The solution required no production at factory_1 and no storage at warehouse_2. Suppose this solution is unacceptable. Now, the production at the two factories can differ by, at most, 100 units. An additional constraint requiring the production at the two factories to be balanced is required. Such a side constraint might look like

```
diff = (  factory_1_warehouse_1
        + factory_1_warehouse_2) -
       (  factory_2_warehouse_1
        - factory_2_warehouse_2)
```

where diff is the name of a nonarc variable that must have values on or between -100 and 100. The value of diff is the amount that factory 1 production exceeds factory 2 production. By default, PROC NETFLOW interprets in side constraint data *from_to* as the flow through the arc directed from node *from* toward node *to* and, although not shown here, you can furnish other arc names in an _name_ variable in the arc data set.

The factory balancing constraint (called balance) is not a part of the network. It is represented, along with the value bounds for diff, in the sparse format in a data

set for side constraints.

```
data side_con;
   input  _type_ $ _row_&$16. _col_&$21.
          _coef_ ;
datalines;
eq  balance  .                          .
.   balance  factory_1_warehouse_1      1
.   balance  factory_1_warehouse_2      1
.   balance  factory_2_warehouse_1     -1
.   balance  factory_2_warehouse_2     -1
.   balance  diff                      -1
lo  lowerbd  diff                    -100
up  upperbd  diff                     100
;
```

To run PROC NETFLOW and display the solution, use

```
proc netflow
   arcdata=network nodedata=nodes
   condata=side_con sparsecondata
   conout=con_sav;
run;
proc print;
   var _from_ _to_ _name_ _cost_ _capac_
       _lo_ _flow_ _fcost_ _rcost_;
   sum _fcost_;
run;
```

The solution is saved in the con_sav data set (Figure 3).

Obs	_from_	_to_	_NAME_	_cost_	_CAPAC_	_LO_	_FLOW_	_FCOST_	_RCOST_
1	warehouse_1	customer_1		3	99999999	0	100	300	.
2	warehouse_2	customer_1		5	99999999	0	0	0	1.0
3	warehouse_1	customer_2		4	99999999	0	75	300	.
4	warehouse_2	customer_2		5	99999999	0	125	625	.
5	warehouse_1	customer_3		4	99999999	0	50	200	.
6	warehouse_2	customer_3		6	99999999	0	0	0	1.0
7	factory_1	warehouse_1		10	99999999	0	0	0	2.0
8	factory_2	warehouse_1		5	99999999	0	225	1125	.
9	factory_1	warehouse_2		7	99999999	0	125	875	.
10	factory_2	warehouse_2		9	99999999	0	0	0	5.0
11			diff	0	100	-100	-100	0	1.5
								=======	
								3425	

Figure 3. PROC NETFLOW solution

Notice that the solution now has production balanced across the factories; the production at factory 2 exceeds that at factory 1 by 100 units.

Network and Linear Programs solved by the IntPoint algorithm

The Simplex algorithm, developed shortly after World War II, was the main method used to solve Linear Programming problems. Over the last decade, the IntPoint algorithm has been developed to also solve LP problems. From the start it showed great theoretical promise, and considerable research in the area resulted in practical implementations that performed competitively with the Simplex algorithm. More recently, IntPoint algorithms have evolved to become superior to the Simplex algorithm, in general, especially when the problems are large. The Primal-Dual Predictor-Corrector IntPoint algorithm has been implemented in PROC NETFLOW.

To solve the Candy problem using the IntPoint algorithm of PROC NETFLOW, specify

```
proc netflow
   arcdata=dense condata=dense;
run;
```

using the dense formatted input data set or, using the equivalent sparse format input dataset, specify

```
proc netflow
   arcdata=sparse
   sparsecondata condata=sparse;
run;
```

The Primal-Dual IntPoint Algorithm

The Linear Program to be solved is

$$min\{c^T x\}$$

subject to $Ax = b$

$$x \geq 0$$

This is the *primal* problem. To simplify the algebra here, assume that variables have zero lower bounds and infinite upper bounds, and all constraints are equalities. (IntPoint algorithms do efficiently handle finite bounds, and it is easy to introduce primal slack variables to change inequalities into equalities.) The problem has n variables. i is a variable number. k is an iteration number, and if used as a subscript or superscript denotes "of iteration k".

There exists an equivalent problem, the *dual* problem, stated as

$$max\{b^T y\}$$

subject to $A^T y + s = c$

$$s \geq 0$$

where y are dual variables, and s are dual constraint slacks

What the IntPoint has to do is solve the system of α equations to satisfy the Karush-Kuhn-Tucker (KKT) conditions for optimality:

$$
\begin{aligned}
&Ax = b \\
&A^T y + s = c \\
&x^T s = 0 \\
&x \geq 0 \\
&s \geq 0
\end{aligned}
$$

These are the conditions for feasibility, with the *complementarity* condition $x^T s = 0$ added. $c^T x = b^T y$ must occur at the optimum. Complementarity forces the optimal objectives of the primal and dual to be equal, $c^T x_{opt} = b^T y_{opt}$, as

$$
\begin{aligned}
&0 = x_{opt}^T s_{opt} = s_{opt}^T x_{opt} = \\
&(c - A^T y_{opt})^T x_{opt} = \\
&c^T x_{opt} - y_{opt}^T (A x_{opt}) = \\
&c^T x_{opt} - b^T y_{opt}
\end{aligned}
$$

$$
0 = c^T x_{opt} - b^T y_{opt}
$$

Before the optimum is reached, a solution $(x y s)$ may not satisfy the KKT conditions.

- Primal constraints can be broken, $infeas_{\,b} = b - Ax \neq 0$.
- Dual constraints can be broken, $infeas_{\,c} = c - A^T y - s \neq 0$.
- Complementarity is unsatisfied, $x^T s = c^T x - b^T y \neq 0$. This is called the *duality gap*.

IntPoint algorithm works by using Newtons method to find a direction to move $(\Delta x^k, \Delta y^k, \Delta s^k)$ from the current solution (x^k, y^k, s^k) toward a better solution.

$$
(x^{k+1}, y^{k+1}, s^{k+1}) = (x^k, y^k, s^k) + \alpha(\Delta x^k, \Delta y^k, \Delta s^k)
$$

is the *step length* and is assigned a value as large as possible but ≤ 1.0 and not so large that a x_i^{k+1} or s_i^{k+1} is "too close" to zero. The direction in which to move is found using

$$
\begin{aligned}
&A\Delta x^k = -infeas_{\,b} \\
&A^T \Delta y^k + \Delta s^k = -infeas_{\,c} \\
&S^k \Delta x^k + X^k \Delta s^k = -X^k S^k e
\end{aligned}
$$

where S = diag(s), X = diag(x), and e = vector with all elements = 1

To greatly improve performance, the third equation is changed to

$$
\begin{aligned}
&S^k \Delta x^k + X^k \Delta s^k = \\
&-X^k S^k e + \sigma_k \mu_k e
\end{aligned}
$$

where $\mu_k = 1/n X^k S^k e$, the average complementarity, and

$$0 \leq \sigma_k \leq 1$$

The effect now is to find a direction in which to move to reduce infeasibilities and to reduce the complementarity toward zero, but if any $x_i^k s_i^k$ is too close to zero, it is "nudged out" to μ, any $x_i^k s_i^k$ that is larger than μ is "nudged into" μ. A σ_k close to or equal to 0.0 biases a direction toward the optimum, and a value for σ_k close to or equal to 1.0 "centers" the direction toward a point where all pairwise products $x_i^k s_i^k = \mu$. Such points make up the *Central Path* in the interior. Although centering directions make little, if any, progress in reducing μ and moving the solution closer to the optimum, substantial progress toward the optimum can usually be made in the next iteration.

The Central Path is crucial to why the IntPoint algorithm is so efficient. This path "guides" the algorithm to the optimum through the interior of feasible space. Without centering, the algorithm would find a series of solutions near each other close to the boundary of feasible space. Step lengths along the direction would be small and many more iterations would probably be required to reach the optimum.

That in a nutshell is the Primal-Dual Interior Point algorithm. Varieties of the algorithm differ in the way α and σ_k are chosen and the direction adjusted each iteration. A wealth of information can be found in the texts by Roos, Terlaky, and Vial 1997, Wright 1996, and Ye 1996.

The calculation of the direction is the most time-consuming step of the IntPoint algorithm. Assume the kth iteration is being performed, so the subscript and superscript k can be dropped from the algebra.

$$A\Delta x = -infeas_b$$
$$A^T\Delta y + \Delta s = -infeas_c$$
$$S\Delta x + X\Delta s = -XSe + \sigma\mu e$$

Rearranging the second equation

$$\Delta s = -infeas_c - A^T\Delta y$$

Rearranging the third equation

$$\Delta s = X^{-1}(-S\Delta x - XSe + \sigma\mu e)$$
$$\Delta s = -\Theta\Delta x - Se + X^{-1}\sigma\mu e$$

where $\Theta = SX^{-1}$

Equating these two expressions for Δs and rearranging

$$-\Theta\Delta x - Se + X^{-1}\sigma\mu e = -infeas_c - A^T\Delta y$$
$$-\Theta\Delta x = Se - X^{-1}\sigma\mu e - infeas_c - A^T\Delta y$$
$$\Delta x = -\Theta^{-1}(-Se + X^{-1}\sigma\mu e + infeas_c + A^T\Delta y)$$

therefore $\Delta x = \rho + \Theta^{-1}A^T\Delta y$

where $\rho = \Theta^{-1}(-Se + X^{-1}\sigma\mu e + infeas_c)$

Substituting into the first direction equation

$$A\Delta x = -infeas_b$$
$$A(\rho + \Theta^{-1}A^T\Delta y) = -infeas_b$$
$$A\Theta^{-1}A^T\Delta y = -infeas_b - A\rho$$
$$\Delta y = (A\Theta^{-1}A^T)^{-1}(-infeas_b - A\rho)$$

Θ, ρ, Δy, Δx and Δs are calculated in that order. The hardest term is the factorization of the $(A\Theta^{-1}A^T)$ matrix to determine Δy. Fortunately, although the *values* of $(A\Theta^{-1}A^T)$ is different each iteration, the *locations* of the nonzeroes in this matrix remain fixed; the nonzero locations is the same as those in the matrix (AA^T). This is due to $\Theta^{-1} = XS^{-1}$ being a diagonal matrix that has the effect of merely scaling the columns of (AA^T).

The fact that the nonzeroes in $A\Theta^{-1}A^T$ has a constant pattern is exploited by all IntPoint algorithms, and is a major reason for their excellent performance. Before iterations begin, AA^T is examined and it's rows and columns permutated so that during Cholesky Factorization, the number of *fillins* created is smaller. A list of arithmetic operations to perform the factorization is saved in concise computer data structures (working with memory locations rather than actual numerical values). This is called *symbolic factorization*. During iterations, when memory has been initialized with numerical values, the operations list is performed sequentially. Determining how the factorization should be performed again and again is unnecessary.

The Primal-Dual Predictor-Corrector IntPoint algorithm

The variant of the IntPoint algorithm implemented in PROC NETFLOW is a Primal-Dual Predictor-Corrector IntPoint algorithm. At first, Newtons method is used to find a direction to move $(\Delta x^k_{aff}, \Delta y^k_{aff}, \Delta s^k_{aff})$, but calculated as if μ is zero, that is, a step with no centering; an *affine* step.

$$A\Delta x^k_{aff} = -infeas_b$$
$$A^T\Delta y^k_{aff} + \Delta s^k_{aff} = -infeas_c$$
$$S^k\Delta x^k_{aff} + X^k\Delta s^k_{aff} = -X^kS^ke$$

$$(x^k_{aff}, y^k_{aff}, s^k_{aff}), y^k, s^k)$$
$$\alpha(\Delta x^k_{aff}, \Delta y^k_{aff}, \Delta s^k_{aff})$$

is the *step length* as before.

Complementarity x^Ts is calculated at $(x^k_{aff}, y^k_{aff}, s^k_{aff})$ and compared with the complementarity at the starting point (x^k, y^k, s^k) and the success of the affine step is gauged. If the affine step was successful in reducing the complementarity by a substantial amount, the need for centering is not great, and the value of σ_k in the following linear system is assigned a value close to zero. If, however, the affine step was unsuccessful, centering would be beneficial, and the value of σ_k in the following linear system is assigned a value closer to 1.0. The value of σ_k is therefore adaptively altered depending on the progress made toward the optimum.

A second linear system is solved to determine a centering vector $(\Delta x^k_c, \Delta y^k_c, \Delta s^k_c)$ from $(x^k_{aff}, y^k_{aff}, s^k_{aff})$

$$A\Delta x^k_c = 0$$
$$A^T\Delta y^k_c + \Delta s^k_c = 0$$
$$S^k\Delta x^k_c + X^k\Delta s^k_c = -X^kS^ke$$
$$S^k\Delta x^k + X^k\Delta s^k =$$
$$-X^k_{aff}S^k_{aff}e + \sigma_k\mu_ke$$

then

$$(\Delta x^k, \Delta y^k, \Delta s^k) = (\Delta x^k_{aff}, \Delta y^k_{aff}, \Delta s^k_{aff}) + (\Delta x^k_c, \Delta y^k_c, \Delta s^k_c)$$

$$(x^{k+1}, y^{k+1}, s^{k+1}) = (x^k, y^k, s^k) + \alpha(\Delta x^k, \Delta y^k, \Delta s^k)$$

where, as before, α is the *step length* assigned a value as large as possible but ≤ 1.0 and not so large that a x_i^{k+1} or s_i^{k+1} is "too close" to zero.

Although the Predictor-Corrector variant entails solving two linear system instead of one, fewer iteration are usually required to reach the optimum. The additional overhead of calculating the second linear system is small as the factorization of the $(A\Theta^{-1}A^T)$ matrix has already been performed to solve the first linear system.

PROC NLP

The NLP procedure (**N**on**L**inear **P**rogramming) offers a set of optimization techniques for minimizing or maximizing a continuous nonlinear function $f(x)$ of n decision variables, $x = x_1, \ldots, x_n)^T$ with lower and upper bound, linear and nonlinear, equality and inequality constraints. This can be expressed as solving

$$\begin{array}{lll} & \min_{x \in \mathcal{R}^n} f(x) & \\ \text{subject to} & c_i(x) = 0 & i = 1, \ldots, m_e \\ & c_i(x) \geq 0 & i = m_e + 1, \ldots, m \\ & u_i \geq x_i \geq l_i & i = 1, \ldots, n \end{array}$$

where f is the objective function, the c_i's are the constraint functions, and u_i, l_i's are the upper and lower bounds. Problems of this type are found in many settings ranging from optimal control to maximum likelihood estimation.

The NLP procedure provides a number of algorithms for solving this problem that take advantage of special structure on the objective function or constraints, or both. One example is the **quadratic programming problem**:

$$\begin{array}{ll} & f(x) = \frac{1}{2}x^T G x + g^T x + b \\ \text{subject to} & c_i(x) = 0 \\ & i = 1, \ldots, m_e \end{array}$$

where the $c_i(x)$'s are linear functions; $g = (g_1, \ldots, g_n)^T$ and $b = b_1, \ldots, b_n)^T$ are vectors; and G is an $n \times n$ symmetric matrix.

Another example is the **least-squares problem**:

$$\begin{array}{ll} & f(x) = \frac{1}{2}\{f_1^2(x) + \cdots + f_l^2(x)\} \\ \text{subject to} & c_i(x) = 0 \\ & i = 1, \ldots, m_e \end{array}$$

where the $c_i(x)$'s are linear functions, and $f_1(x), \ldots, f_m(x)$ are nonlinear functions of x.

The following optimization techniques are supported in PROC NLP:

- Quadratic Active Set Technique
- Trust-Region Method
- Newton-Raphson Method With Line Search
- Newton-Raphson Method With Ridging
- Quasi-Newton Methods
- Double-Dogleg Method
- Conjugate Gradient Methods
- Nelder-Mead Simplex Method (NMSIMP)
- Levenberg-Marquardt Method
- Hybrid Quasi-Newton Methods

These optimization techniques require a continuous objective function f, and all but one (NMSIMP) require continuous first-order derivatives of the objective function f. Some of the techniques also require continuous second-order derivatives. There are three ways to compute derivatives in PROC NLP:

1. analytically (using a special derivative compiler), the default method
2. via finite difference approximations
3. via user-supplied exact or approximate numerical functions

Nonlinear programs can be input into the procedure in various ways. The objective, constraint, and derivative fucntions are specified using the programming statements of PROC NLP. In addition, information in SAS data sets can be used to define the structure of objectives and constraints as well as specify constants used in objectives, constraints, and derivatives.

PROC NLP uses data sets to input various pieces of information.

- The DATA= data set enables you to specify data shared by all functions involved in a least squares problem.

- The INQUAD= data set contains the arrays appearing in a quadratic programming problem.
- The INVAR= data set specifies initial values for the decision variables, the values of constants that are referred to in the program statements, and simple boundary and general linear constraints.
- The MODEL= data set specifies a model (functions, constraints, derivatives) saved at a previous execution of the NLP procedure.

PROC NLP uses data sets to output various results.

- The OUTVAR= data set saves the values of the decision variables, the derivatives, the solution, and the covariance matrix at the solution.
- The OUT= output data set contains variables generated in the program statements defining the objective function as well as selected variables of the DATA= input data set, if available.
- The OUTMODEL= data set saves the programming statements. It can be used to input a model in the MODEL= input data set.

As an alternative to supplying data in SAS data sets, some or all data for the model can be specified using SAS programming statements. These are similar to those used in the SAS DATA step.

Consider the simple example of minimizing the Rosenbrock Function (Rosenbrock, 1960).

$$f(x) = \frac{1}{2}\{100(x_2 - x_1^2)^2 + (1 - x_1)^2\} = \frac{1}{2}\{f_1^2(x) + f_2^2(x)\}, \ = (x_1, x_2)$$

The minimum function value is $f(x^*)$ at $x^* = (1, 1)$. This problem does not have any constraints.

The following PROC NLP run be used to solve this problem:

```
proc nlp;
   min f;
   decvar x1 x2;
   f1 = 10 * (x2 - x1 * x1);
   f2 = 1 - x1;
   f  = .5 * (f1 * f1 + f2 * f2);
run;
```

The MIN statement identifies the symbol f that characterizes the objective function in terms of $f1$ and $f2$, and the DECVAR statement names the decision variables $X1$ and $X2$. Because there is no explicit optimizing algorithm option specified (TECH=), PROC NLP would use the Newton-Raphson method with ridging, the default algorithm when there are no constraints.

A better way to solve this problem is to take advantage of the fact that f is a sum of squares of $f1$ and $f2$ and to treat it as a least-squares problem. Using the LSQ statement instead of the MIN statement tells the procedure that this is a least-squares problem, which results in the use of one of the specialized algorithms for solving least-squares problems (for example, Levenberg-Marquardt).

```
proc nlp;
   lsq f1 f2;
   decvar x1 x2;
   f1 = 10 * (x2 - x1 * x1);
   f2 = 1 - x1;
run;
```

The LSQ statement results in the minimization of a function that is the sum of squares of functions that appear in the LSQ statement.

The least-squares specification is preferred because it enables the procedure to exploit the structure in the problem for numeric stability and performance.

There are several other NLP statements that are used to supply additional data of the model, such as variable value bounds and linear and nonlinear constraints. The following is an example of a problem with bounds and with linear and nonlinear constraints:

```
proc nlp tech=QUANEW;
   min f;
   decvar x1 x2;
   bounds x1 - x2 <= .5;
   lincon x1 + x2 <= .6;
   nlincon c1 >= 0;

   c1 = x1 * x1 - 2 * x2;

   f1 = 10 * (x2 - x1 * x1);
   f2 = 1 - x1;

   f = .5 * (f1 * f1 + f2 * f2);
run;
```

PROC TRANS

Transportation networks are a special type of network, called *bipartite* networks, that have only supply and demand nodes and arcs directed from supply nodes to demand nodes. For these networks, data can be given most efficiently in a rectangular or matrix form. The TRANS procedure takes cost, capacity, and lower bound data in this form. The observations in these

data sets correspond to supply nodes, and the variables correspond to demand nodes.

```
data transprt;
   input source $ supply cust_1 cust_2 cust_3 ;
   datalines;
demand       .   100    200     50
factory1   500    10      9      7
factory2   500     9     10      8
;
```

This data set shows the source names as the values for the source variable, the supply at each source node as the values for the supply variable, and the unit shipping cost for source to sink as the values for the sink variables cust_1 to cust_3. Notice that the first record contains the demands at each of the sink nodes.

The TRANS procedure finds the minimum cost routing. It solves the problem and saves the solution in an output data set.

```
proc trans
   nothrunet data=transprt out=transout;
   supply supply;
   id source;
run;
proc print;
run;
```

The optimum solution total (3050) is reported on the SAS log. The solution (Figure 4) shows the amount to ship from each factory to each customer, per day.

Obs	source	supply	cust_1	cust_2	cust_3	_DUAL_
1	_DEMAND_	.	100	200	50	.
2	factory1	500	0	200	50	0
3	factory2	500	100	0	0	0
4	_DUAL_	.	9	9	7	.

Figure 4. PROC TRANS solution

PROC ASSIGN

The assignment problem is a special type of transportation problem, one having supply and demand values of one unit. As with the transportation problem, the cost data for this type of problem are saved in a SAS data set in rectangular form.

Improvements for PROC NETFLOW

PROC NETFLOW was originally designed to solve Network Programming problems with side constraints. Now it can solve LPs as well. This is automatically detected when while reading data.

CPU Time Reductions

To solve LPs, PROC NETFLOW uses an IntPoint algorithm. Compared to the Simplex algorithm employed by PROC LP, IntPoint has been developed over only the last decade. It has been described in the literature as being superior, particularly when solving large LPs. After implementing the IntPoint, this is our experience as well.

Available to the Operations Research community is a set of about 100 LPs collected over the years as being large (by the standard of the time), difficult to solve, or representative of real-world problems. We use these to test PROC LP and now PROC NETFLOW. Many of these LPs have several thousand variables and constraints. See Figure 5 in the Appendix for details of the test problems sizes and solution times for both the IntPoint algorithm and PROC LP.

When we compare the the performance of the IntPoint algorithm with that of the Simplex algorithm, after first ordering the problems by nettime divided by lptime, we see that

- IntPoint solves 6 problems that Simplex has trouble with, or takes a very long time.
- IntPoint solves the next quarter of the problems 10 or more times faster than Simplex. Furthermore, these are among the largest problems. For 21 problems for which we have solution times for both IntPoint and Simplex, IntPoint takes a total of 17 minutes, Simplex 8 hours. Average 50 seconds versus 23 minutes
- IntPoint solves the next two thirds of the problems 2.5 times or better faster than Simplex. These problems are some of the smaller ones.
- Only about 10 percent of the problems were solved faster by Simplex than by IntPoint. However, these were among the very smallest problems. For the 12 problems for which we have solution times for both IntPoint and Simplex, IntPoint takes a total of 71 seconds, Simplex 34 seconds. Average 5.5 seconds versus 3 seconds
- Simplex solves 1 problem that IntPoint has trouble with. The main matrix calculated during each iteration of the IntPoint algorithm becomes numerically unstable as the optimum is approached; ways to remedy this are under investigation.

One especially appealing feature of the IntPoint algorithm is it's ability to tackle very large LP problems. PROC NETFLOW was able to read the data for an LP with 1.3 million variables and 300,000 constraints, determine the optimum, and create an output dataset containing the solution in under 2 hours on a PC. (Incidently, this is a real-world model, the result of collaboration between staff members of the Institute and

a company that refines and distributes sugar. (Attend next years SUGI to see a paper on this project!) By increasing the number of time periods, an LP with 1.7 million variables and 400,000 constraints was created. PROC NETFLOW took only a few minutes more to solve that problem. This problem was still able to solved when we instructed PROC NETFLOW to use no more than 500 Mbytes of working memory.

Improvements for PROC LP

A considerable amount of effort has gone into making the Integer programming facilities of PROC LP more efficient, both in terms of CPU time and memory required.

Better Sort Routines while Branching and Bounding

When solving an integer programming problem, PROC LP divides the original problem into many subproblems. At each iteration, PROC LP tries to solve one of the subproblems. If the solution has a noninteger value assigned to a variable that must have an integer value, PROC LP further divides the subproblem into two subproblems. This is called *branching and bounding*. As the number of iterations increases, the number of subproblems that need to be solved can increase dramatically. In the worst case, the number of subproblems can equal the number of iterations. Several test problems are so large that over a million iterations are executed.

There are several list data structures that order the desirability of selecting each subproblem to have branching occur from. When a new subproblem is generated or a subproblem is removed, the lists need to be updated. To update each list at each iteration, sorting is needed. Experiments were conducted that found that sometimes it was the sorting that took a significant portion of the CPU time in PROC LP. Special sorting routine were implemented that improved performance on all of our test problems. Of 36 large real-world problems. the CPU time used to solve 13 of them decreased by over 15 percent. For the larger problems, the solution times were more significant; sometimes a third or less. The magnitude of the improvements depend on the average number of the active subproblems generated by PROC LP during the iteration. The larger the number is, the greater the improvement.

Memory Usage Reduction

The core information of each subproblem is saved in memory. When the number of subproblems increase to when an Operating System starts to use the virtual memory, the performance of PROC LP drops dramatically. By consolidating the core information, which reduced by 10 percent memory usage on each subproblem, the performance of PROC LP improved.

The number of integer variables in some of the test problems can number several hundred. Some of the information about each subproblem that is saved in a utility file on disk includes lower and upper bounds of integer variables. As the number of subproblems increases, the size of the disk utility file quickly increases. In one testing problem, the utility file size grew to over 400 Mbytes during the iterations. However, in most of the integer programming problems, all integer variables are binary variables, which means their lower and upper bounds are either 0 or 1. By changing the way bound information is stored for binary variables, (using bit manipulation instead of byte arrays), the overall storage usage were reduced by over 50 percent. Solution times were reduced by having to exchange less data between memory and disk.

Preprocessing

Preprocessing, which occurs before optimization commences, tries to reduce model sizes by identifying redundant constraints, or by *fixing* variables to their optimal values by making deductions that conclude those variable can have only those values assigned to them and no others. Redundant constraints and fixed variables can be dropped from the eventual model actually solved. Preprocessing is often so successful that the size of the problem to be solved is half or less the size of the original problem. (Incidently, preprocessing is performed with equal success by PROC NETFLOW prior to IntPoint optimization.)

For integer programming problems, preprocessing can often reduce the gap between an integer program and it's relaxed LP, which will likely lead to a reduced branch and bound tree. During our testing, this significiantly improved the solution times for most problems and allowed PROC LP to solve several problems that could not be solved previously.

References

Roos, C. Terlaky, T. Vial, J.-Ph. (1997) "Theory and Algorithms for Linear Optimization", John Wiley & Sons.

Wright, S.J. (1996) "Primal-Dual Interior Point Algorithms", Philadelphia: SIAM

Ye, Y. (1996) "Interior Point Algorithms: Theory and Analysis", Wiley-Interscience Series in Discrete Mathematics and Optimization.

AUTHOR

Trevor D. Kearney, SAS Institute Inc., SAS Campus Drive, Cary, NC 27513. FAX (919) 677-4444 Email sastdk@wnt.sas.com

Appendix: LP Test Problems

The following table in Figure 5 contain the size and solution times for the test LP problems. The values of the nvar, nle, neq, nge, and ncoefs are the number of variables, the number of $\leq$ constraints, the number of $=$ constraints, the number of $\geq$ constraints, and the number of constraint coefficients in them, respectively. The values of nettime and lptime are the CPU times used by PROC NETFLOW (using the Int-Point algorithm) and PROC LP, respectively, with format *minutes:seconds.hundredth of seconds*, run on a HP 715/75 workstation (which is slower than today's PCs).

Obs	problem	nvars	nle	neq	nge	ncoefs	nettime	lptime
1	adlittle	97	40	15	1	383	0.99	1.33
2	afiro	32	19	8	0	83	0.76	0.45
3	agg	163	405	36	47	2410	6.26	3.05
4	agg2	302	456	60	0	4284	12.11	4.14
5	agg3	302	456	60	0	4300	12.57	4.00
6	bandm	472	0	305	0	2494	3.24	21.60
7	beaconfd	262	33	140	0	3375	2.34	2.23
8	bore3d	315	19	214	0	1429	2.05	3.33
9	brandy	249	43	139	0	2148	2.78	10.93
10	capri	355	75	142	54	1767	3.87	7.92
11	czprob	3523	37	890	0	10669	10.15	1:33.28
12	e226	282	185	33	5	2578	3.41	8.33
13	etamacro	688	48	272	80	2409	8.06	14.08
14	fffff800	854	93	350	81	6227	12.90	27.76
15	forplan	422	24	90	21	4564	8.28	9.84
16	ganges	1681	25	1284	0	6912	16.92	55.17
17	gfrdpnc	1092	68	548	0	2377	3.55	17.98
18	grow15	892	0	300	0	5620	5.15	26.49
19	grow22	1774	0	440	0	8252	10.19	54.78
20	grow7	301	0	140	0	2612	2.85	7.82
21	israel	142	174	0	0	2269	9.16	4.61
22	nesm	2924	182	480	0	13288	24.87	3:21.20
23	pilotja	1988	151	661	128	14698	1:05.68	26:24.41
24	pilotwe	2789	30	583	109	9126	22.06	10:43.30
25	pilot	3652	1191	233	17	43167	10:51.80	long time
26	pilotnov	2172	151	701	123	13057	1:29.16	9:20.22
27	recipe	180	6	67	18	663	1.74	1.39
28	sc205	203	113	91	0	551	1.81	3.80
29	scagr25	500	146	300	25	1554	2.41	21.63
30	scagr7	140	38	84	7	420	1.65	2.36
31	scfxm1	457	143	187	0	2589	3.52	10.07
32	scfxm2	914	286	374	0	5183	6.90	31.06
33	scfxm3	1371	429	561	0	7777	9.84	1:04.71
34	scorpion	358	48	280	60	1426	2.24	7.08
35	scrs8	1169	59	384	47	3182	4.95	26.83
36	scsd1	760	0	77	0	2388	2.33	9.37
37	scsd6	1350	0	147	0	4316	3.16	19.61
38	scsd8	2750	0	397	0	8584	5.19	1:25.54
39	sctap1	480	0	120	180	1692	2.13	7.43
40	sctap2	1880	0	470	620	6714	8.89	34.66
41	sctap3	2480	0	620	860	8874	12.45	53.94
42	seba	1029	0	507	8	4359	20.73	8.80
43	share1b	225	28	89	0	1151	2.20	5.69
44	share2b	79	83	13	0	694	1.71	2.30
45	shell	1775	2	534	0	3556	4.19	24.22
46	ship04l	2118	40	312	8	6332	4.15	19.71
47	ship04s	1458	40	312	8	4352	3.21	13.24
48	ship08s	2387	72	632	8	7114	4.08	32.11
49	ship08l	4283	72	632	8	12802	7.77	57.76
50	ship12s	2763	101	936	5	8178	5.64	1:00.43
51	ship12l	5427	101	936	5	16170	10.85	1:54.70
52	sierra	2036	633	528	66	7302	10.49	39.37
53	stair	467	147	209	0	3856	7.23	31.22
54	standata	1075	199	160	0	3031	3.24	5.20
55	standmps	1075	199	268	0	3679	4.46	12.56
56	stocfor1	111	48	63	6	447	1.69	1.71
57	stocfor2	2031	888	1143	126	8343	14.85	2:19.23
58	vtpbase	203	133	55	10	908	1.51	7.03
59	z25fv47	1571	305	515	0	10400	28.25	8:54.09
60	z80bau3b	9799	33	0	2204	21002	1:08.06	16:39.67
61	greenbea	5405	107	2196	86	30877	1:10.37	32:20.14
62	greenbeb	5405	107	2196	86	30877	55.14	26:06.86
63	willett	494	94	90	0	2309	4.86	20.23
64	blend	83	31	43	0	491	1.62	1.94
65	bnl1	1175	195	231	206	5121	9.41	1:29.21
66	bnl2	3489	438	1327	515	13999	3:02.93	13:08.14
67	boeing1	385	93	9	246	3574	5.45	16.97
68	boeing2	144	20	4	116	1215	2.06	3.18
69	cycle	2857	146	1376	364	20720	1:36.32	4:03.44
70	d2q06c	5167	664	1507	0	32417	5:17.13	55:39.88
71	degen2	534	223	221	0	3978	14.96	45.28
72	degen3	1818	786	717	0	24646	6:37.60	21:32.25
73	dfl001	12231	0	6071	0	37293	29.30	bad solutn
74	finnis	614	302	47	148	2310	5.12	18.39
75	fit1d	1026	12	1	11	13404	7.81	29.39
76	fit1p	1677	0	627	0	9868	22.10	1:24.04
77	fit2d	10500	10	1	14	129018	1:12.33	long time
78	fit2p	13525	0	3000	0	50284	3:02.04	long time
79	kb2	41	12	16	15	286	2.61	1.82
80	lotfi	308	42	95	16	1078	2.72	bad solutn
81	maros	1443	398	323	124	9614	19.46	2:31.56
82	perold	1376	40	495	90	6018	30.07	5:22.50
83	sc105	103	59	45	0	280	1.37	1.47
84	sc50a	48	29	20	0	130	0.84	0.72
85	sc50b	48	28	20	0	118	0.82	0.74
86	tuff	587	15	261	18	4520	7.66	4:59.76
87	wood1p	2594	0	243	1	70215	39.55	1:51.81
88	woodw	8405	9	1085	4	37474	38.84	4:47.88
89	truss	8806	0	1000	0	27836	44.79	long time
90	ken07	3602	0	2426	0	8404	10.26	4:16.91
91	pds02	7535	181	2772	0	16390	38.39	5:18.30
92	crea	4067	2877	247	304	14987	28.88	9:36.58
93	crec	3678	2420	253	313	13244	24.62	30: approx
94	stocfor3	15695	6866	8829	980	64875	4:06.56	4hrs approx

Figure 5. LP Test Problems