Paper 011-2007

SAS® And Sudoku

Richard A. DeVenezia, Independent Consultant
John R. Gerlach, MaxisIT, Inc.
Larry Hoyle, Institute for Policy and Social Research, Univ. of Kansas
Talbot M. Katz, Analytic Data Information Technologies
Rick Langston, SAS Institute Inc.

ABSTRACT

Sudoku puzzles were first popularized in Japan and have taken the world by storm. The puzzle consists of 81 cells arranged in a 9x9 grid, that is broken into 9 3x3 grids. Given a partially filled grid, the solver must fill in the grid so that every row, every column, and every 3x3 box contains the digits 1 through 9.

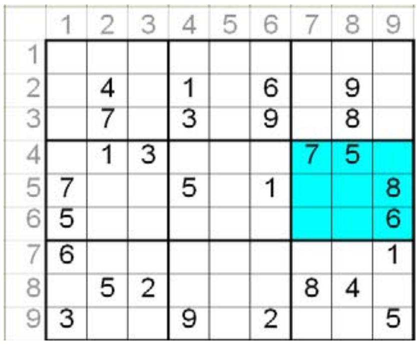
Sudoku solvers can be programmed in the SAS System using a surprising number of methods. This panel discussion will present several approaches to solving Sudoku puzzles, the use of analytic procedures like PROC CLP and PROC LP, several Data Step approaches, and an SQL based approach. Issues that will be addressed include solution speed, ability to solve all possible puzzles, validation of puzzles, specification of puzzles - the effect of using fewer sufficient rules for the game, generation of puzzles, automating the rating of the difficulty of puzzles.

INTRODUCTION

The spell checker in the predominant word processing program suggests “Seduce” when “Sudoku” is typed. Not just spell checkers find Sudoku seductive. There are many books and web sites offering Sudoku puzzles and discussing strategies for solving Sudoku puzzles. Sudoku has also attracted the interest of mathematicians. A *Scientific American* article (Delahaye) gives a good overview of the mathematics of Sudoku.

It was only natural then, that those with access to SAS would turn to it to implement techniques for solving these puzzles. The diversity of those solutions illuminates the range of tools available in the SAS system.

Several approaches have already been published. One early approach used SAS to launch an external solving procedure (Matthews). Another used PROC LP (Kruger). Two others used data step approaches (Gerlach)(Karwe et al). This panel will discuss several other approaches.



	1	2	3	4	5	6	7	8	9
1									
2		4		1		6		9	
3		7		3		9		8	
4		1	3				7	5	
5	7			5		1			8
6	5								6
7	6								1
8		5	2				8	4	
9	3			9		2			5

The rules for Sudoku are simple. A basic puzzle is a 9 by 9 grid. The numbers 1 through 9 can appear in each cell of the grid. Each of the 9 values must appear only once in its row, column and 3 by 3 block (usually marked by heavier lines). A valid puzzle is presented with enough of the values filled in to guarantee a unique solution. Consider the puzzle at the right. Cell (5,8), the cell at row 5 and column 8, cannot have the 1, 5, 7, or 8 that are in its row. It cannot have the 4, 5, 8, or 9 from its column and it cannot have the 5, 6, 7, or 8 from its 3 by 3 block. These constraints together leave only the possibility of a 2 or a 3 for that cell.

There are many strategies for finding values. Some directly find new values. Cell (9,2) at the right, for example, must contain an 8 since it is the only unconstrained value. Other strategies work to increase the set of numbers that are known to be impossible for some of the empty cells. When all other strategies fail it is possible to solve a puzzle by guessing at a value for a cell and trying to solve the puzzle from that point. If that guess leads to no solution then additional guesses are made for other empty cells. If this sequence of guesses leads to a dead end, then the technique “backs up” to the last guessed value and guesses a different value. This backtracking, or trial and error, procedure can always find a solution if one exists, but could be unbearably tedious as a pencil and paper strategy. For this reason puzzles that require backtracking are sometimes referred to as “unfair”.

In what follows, each of the panelists describes one or more approaches to automating solving Sudoku puzzles.

It was the middle of May 2006 when SAS-L was abuzz with a rash of Sudoku puzzle solving. Not being one to avoid a programming challenge I jumped in and posted a reply to "SUDOKU via CLP". It begins:

> *Here is a one step DATA Step Sudoku solver. It's a clean room algorithm, as I have not read any of the articles or links previously posted. It has not been tested for impossible or misentered puzzles. In fact, it has only been run against Larry Hoyle's sample puzzle.*
>
> *I call the algorithm the 'left handed rat finds cheese in an 81 dimensional trap". Its a relatively simple heuristic that does a 'one step onesy heads up' deterministic walk through the possibilities. The path of the walk is left to right, top to bottom. The algorithm could be improved by extending the heads up from a 'one step onesy' to a 'follow the onesies'.*

The complete post can be retrieved from the SAS-L archives or groups.google.com. The program makes no attempt to utilize the logic of elimination that is inherent in Sudoku. Instead, a fixed path through the initial missing values is created (that's the deterministic walk part) and digits are iteratively tried at each succeeding position of the path. When the Sudoku rules are violated the next digit is tried. If there is no next digit to try, the path is shortened and a next digit is tried. No attempt was made to prove the algorithm was or was not defeatable. In other words, there might be initial conditions that cause the program to give up, when, in fact, there is a valid solution.

A path searching algorithm works with several pieces; data structures for geometry and state, rules for valid state, and rules for next step. In general, good bookkeeping is needed to know where you are and how you got there.

INITIAL CONDITIONS AND BOOKKEEPING

The initial conditions of the puzzle are encoded in an 81 character string. Digits are present for known placements and dot (.) indicating the locations of unknowns.

"{ row 1 } { row 2 } … { row 9 }"

```
..........4.1.6.9..7.3.9.8..13...75.7..5.1..85.......66.......1.52...84.3..9.2..5
```

Entering the puzzle in a datalines section makes the geometry obvious.

```
data puzzles;
  input …; puzzleString = …;
datalines;
.........
.4.1.6.9.
.7.3.9.8.
.13...75.
7..5.1..8
5.......6
6.......1
.52...84.
3..9.2..5
#
```

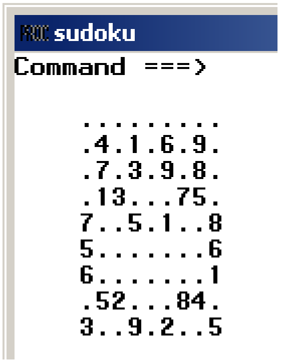
A two dimensional array (9x9) named **grid** is filled in based on the puzzle string.

```
i = 0;

do row = 1 to 9;
do col = 1 to 9;
  i + 1;
  grid [ row, col ]
  = input (substr(puzzleString,i), 1.);
end;
end;

put (grid[*]) (9*2. /) /;
```

```
 --- LOG ---
. . . . . . . . .
. 4 . 1 . 6 . 9 .
. 7 . 3 . 9 . 8 .
. 1 3 . . . 7 5 .
7 . . 5 . 1 . . 8
5 . . . . . . . 6
6 . . . . . . . 1
. 5 2 . . . 8 4 .
3 . . 9 . 2 . . 5
```

The elements of the **grid** array are variables **cell_1** to **cell_81**. The **cell:** variables are used in a DATA Step window definition. The window can be displayed whenever the state of the solution needs to be presented.

```
%macro swin;
    rows=21 columns=60
    %let i = 1;
    %do row = 1 %to 9;
    %do col = 1 %to 9;
      #(1+&row)@(4+&col) cell_&i 1. protect=yes
      %let i = %eval (&i+1);
    %end;
    %end;
  %mend;

  DATA _null_ ;
    window sudoku %swin;
…    display sudoku;
```



```
sudoku
Command ===>

     .........
     .4.1.6.9.
     .7.3.9.8.
     .13...75.
     7..5.1..8
     5.......6
     6.......1
     .52...84.
     3..9.2..5
```

GARBAGE IN ?
It would be folly to try to solve a puzzle if the initial conditions are invalid. A loopy approach would, for each input cell of grid, expressly iterate over the row, column and block of the cell while looking for duplicates. The following instead uses a binary value to track whether or not a digit has been previously observed in a row, column or block.

The entity array maintains a bit value for the digits of each row, column and block. For each digit in the grid a mask is constructed. If the mask, binary ANDed with the entity, is non-zero, then the digit has been previously observed and thus puzzle string is invalid because of a repeat in a row, column or block. Otherwise the mask is binary ORed with the entity to turn on the bit corresponding to the digit.

```
  * validate the puzzle;

  array entity [9,3] (27*0);

  do row = 1 to 9;
  do col = 1 to 9;
    value = grid [row,col];
    if value = . then CONTINUE;

    mask = BLSHIFT(1,value);

    if BAND(entity [row,1],mask) ne 0 then do;
      put 'ERROR: BOGUS puzzle row found @ ' row= col= value=;
      stop;
    end;

    entity [row,1] = BOR (entity [row,1], mask);

    if BAND(entity [col,2],mask) ne 0 then do;
      put 'ERROR: BOGUS puzzle column found @ ' row= col= value=;
      stop;
    end;

    entity [col,2] = BOR (entity [col,2], mask);

    block = 1 + floor ((row-1) / 3) * 3 + floor ((col-1) / 3);

    if BAND(entity [block,3],mask) ne 0 then do;
      put 'ERROR: BOGUS puzzle block found @ ' row= col= value=;
      stop;
    end;

    entity [block,3] = BOR (entity [block,3], mask);
```

PATH
A fixed path through the missing values is determined. The simplest path is left to right, top to bottom. A two dimensional support array (81x4) named **`empty`** is used to track the coordinates of missing values in the **`grid`** and the coordinates of the values block. If there are *N* digits in the puzzle string, then the first 81-*N* (MaxEmpty) slots in **`empty`** will be utilized. What values are associated with the second dimensions indices?

I,1 - row of the ith missing
I,2 - column of the ith missing
I,3 - upper left row of block containing the ith missing
I,4 - upper left column of block containing the ith missing

```
pathOfLeftToRightTopToBottom:
  i = 0;
  do row = 1 to 9;
  do col = 1 to 9;
    if grid[row,col] > 0 then CONTINUE;
    i + 1;
    empty[i,1] = row;
    empty[i,2] = col;
    empty[i,3] = 1 + 3 * floor ((row-1)/3);
    empty[i,4] = 1 + 3 * floor ((col-1)/3);
  end;
  end;
  maxempty = i;
return;
```

A two dimensional array (9x9) named **`last`** is used to track the last tried digit at each grid coordinate.

CHOICES
Three 9x10 arrays, **`rowChoices`**, **`colChoices`** and **`blkChoices`**, keep track of the available digit choices for each row, column and 3x3 block. A usage count is maintained in the 10th element . Initially,

```
rowChoices [row,digit] = 1;    * row and digit 1..9;
colChoices [col,digit] = 1;    * row and digit 1..9;
blkChoices [blk,digit] = 1;    * blk and digit 1..9;
```

A 9x9 support matrix **`blkOfCell`** is used to map a grid coordinate to a block. The use of a mapping array is an optimizing technique for reducing the number of math operations at the interior of the search.

```
blkOfCell [row,col] = 1 + 3 * floor ((row-1) / 3) + floor ((col-1) / 3);
```

The choice arrays are updated and the empty array populated during an initial scan over **`grid`**. The variable **`p`** represents digit and means position in an array of nine distinct items. Sudoku traditionally focuses on the digits 1 to 9. It could equally deal with nine musical notes, or nine varieties of wine.

```
i = 0;
do row = 1 to 9;
  do col = 1 to 9;
    blk = blkOfCell [ row, col ];
    p = grid [ row, col ];
    if p then do;
      rowChoices { row, p } = .; rowChoices { row, 10 } + 1;
      colChoices { col, p } = .; colChoices { col, 10 } + 1;
      blkChoices [ blk, p ] = .; blkChoices [ blk, 10 ] + 1;
    end;
  end;
end;
```

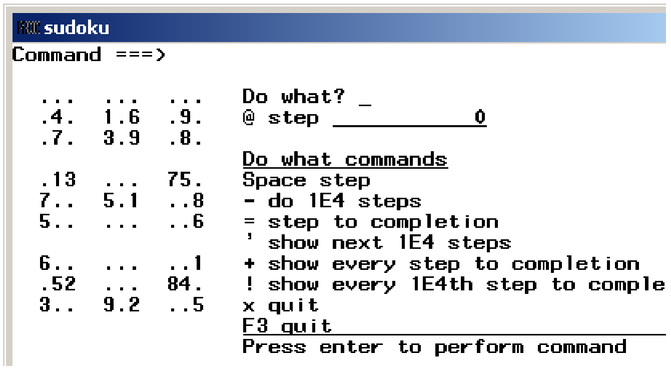
Later, when a candidate digit **p** is placed in the grid, the following bookkeeping is done:

```
grid[row,col] = p;
blk = blkOfCell [ row, col ];

rowChoices[row,p] = -rowChoices[row,p];
colChoices[col,p] = -colChoices[col,p];
blkChoices[blk,p] = -blkChoices[blk,p];

rowChoices[row,10] + 1;
colChoices[col,10] + 1;
blkChoices[col,10] + 1;
```

A negative choice indicates a digit has been temporarily allocated, and thus can not be a candidate in future considerations. The +1 operation increments the usage count for row and column in which the digit is placed. Note: when usage count reaches 8, the remaining digit placement becomes a given (if feasible).

STEPS AND CONTROL SURFACE

The step is a mid-level interior operation of the search algorithm. The step code is called using a LINK statement. This affords the opportunity to interact with the user prior to actually performing the next step.



```
sudoku
Command ===>

 ...  ...  ...   Do what? _
 .4.  1.6  .9.   @ step ___________0
 .7.  3.9  .8.
                 Do what commands
 .13  ...  75.   Space step
 7..  5.1  ..8   - do 1E4 steps
 5..  ...  ..6   = step to completion
                 ' show next 1E4 steps
 6..  ...  ..1   + show every step to completion
 .52  ...  84.   ! show every 1E4th step to comple
 3..  9.2  ..5   x quit
                 F3 quit
                 Press enter to perform command
```

When a step, or set of step operations completes, a dialog is presented to the user showing the current state of the search and a menu of "Do what commands" that let the user indicate how many and of what nature the next steps should be.

When code flow returns from step, the variable **`index`** will contain a missing value if there is nothing left to try -- meaning the puzzle is solved or no solution was found.

Show each of the next 10,000 steps

```
if exit = "'" then
do stepi = 1 to 1e4
  while (index ne .);
  link step;
  display sudoku noinput;
end;
```

Show every step to completion (noinput means that the user will never be able to change **`exit`** from '+')

```
else
if exit = '+' then
do until (index=.);
  link step;
  display sudoku noinput;
end;
```

Show every 10,000th step on way to completion. Again, noipnut means ! can not be changed.

```
else
if exit = '!' then
do until (index=.);
  do stepi = 1 to 1e4
  while (index ne .);
    link step;
  end;
  display sudoku noinput;
end;
```

Step to completion, without showing anything else.

```
else
if exit = '=' or autoRunToCompletion
then
do until (index=.);
  link step;
end;
```

Perform 10,000 steps quietly -- no displays

```
else
if exit = '-' then
do stepi = 1 to 1e4;
  link step;
end;
```

STEP MECHANICS
The operation of a step is to place a value that does not violate Sudoku rules. The rules for placement are:
- follow the fixed path (increment **`index`**)
- loop until the next higher digit conforms
- backup if 9 does not fit (decrement **`index`**)

follow the path

```
step + 1;
index + 1;

row = empty [ index, 1 ];
col = empty [ index, 2 ];
blk = blkOfCell[row,col];
```

loop until next digit conforms

```
do until
  (rowChoices[row,p] > 0 and
   colChoices[col,p] > 0 and
   blkChoices[blk,p] > 0
  );

  last[row,col]+1;

  p = last[row,col];

  if p > 9 then do;
    * dead end processing;
  end;
end; * do until;

* corral conforming digit, tag choice as in use and update grid;

rowChoices[row,p] = -1;
colChoices[col,p] = -1;
blkChoices[blk,p] = -1;

rowChoices[row,10] + 1;
colChoices[col,10] + 1;
blkChoices[blk,10] + 1;

grid[row,col] = p;
```

backup when dead-ended

```
 last[row,col] = .;

 * back off and release;

 index + (-1);

 row = empty [ index, 1 ];
 col = empty [ index, 2 ];

 blk = blkOfCell[row,col];
 p = last[row,col];

 rowChoices[row,p] = +1;
 colChoices[col,p] = +1;
 blkChoices[blk,p] = +1;

 rowChoices[row,10] + (-1);
 colChoices[col,10] + (-1);
 blkChoices[blk,10] + (-1);

 grid[row,col] = .; * for cosmetic purposes when updateViewer called;
```

```
    index + (-1);
    return;
```

CONCLUSION
Programming a Sudoku solver that uses a fixed path approach was an interesting challenge. SAS Data Step demonstrates itself as a very efficient machine for array processing. Additionally, SAS-L can be a great distraction ☺

GERLACH – USING A CUBE TO SOLVE THE SQUARE

Imagine a cube positioned behind the Sudoku square. The possible values in the square can be represented as a triple (x,y,z) such that X and Y represent the row/column coordinate and Z represents the vector of possible values at the XY coordinate. In fact, the Z coordinate *is* the possible value for the XY coordinate, that cell in the Sudoku Square. Thus, the cube represents a collection of orthogonal vectors containing the possible values for each cell in the grid. There are two more important observations about the cube:

- There are n^2 triples (i.e., the answer to the puzzle) of the form *x,y,z* where $1 \le x,y,z \le n$.
- All *XY* pairs are different, all *XZ* pairs are different, and all *YZ* pairs are different.

Keep in mind that the cube contains the possible values associated with each blank cell in the square.

The proposed SAS solution emulates the notion of constraint propagation; meaning, the event of placing a number in a cell invokes the process of updating, in this case, an abstract data structure (the cube) in order to identify viable numbers for other cells. Most programming solutions use the backtracking method, which is little more than the trial and error method relying on the power of the computer and special procedures afforded by SAS.

ELEMENTS OF THE SQUARE AND CUBE

In the SAS program, the Sudoku square is represented by the set of variables having the form XY*xy*, where *xy* denotes the coordinate location in the square. Thus, for a 9x9 matrix, there are 81 variables named:

XY11-XY19; XY21-XY29;…; XY91-XY99

Similarly, the cube is is represented by the set of variables having the form XYZ*xyz*, where *xyz* denotes the coordinate location of the cell in the cube. Thus, the cube contains 729 variables named:

XYZ111-XYZ119; XYZ191-XYZ199; …; XYZ911-XYZ919; XYZ991-XYZ999

So, for example, the variable XYZ258 represents the 2nd row, the 5th column, and the 8th cell in cube. Again, notice that the Z coordinate denotes the possible value that is assigned to the XY coordinate of the square, the value 8.

The *%elements* macro below generates the enumerated variable names for the square and the cube by specifying the geometric figure, either a square or a cube. Notice that the macro takes advantage of the abbreviated form of enumerated variables in the SAS language by using the dash (e.g., XY11-XY19).

```
%macro elements(type);
   %if %upcase(&type.) ne CUBE
      %then %do;
         %do x = 1 %to 9; xy&x.1-xy&x.9 %end;
         %end;
      %else %do;
         %do x = 1 %to 9;
            %do y = 1 %to 9; xyz&x.&y.1-xyz&x.&y.9 %end;
            %end;
         %end;
%mend elements;

%elements(square) ;

%elements(cube)   ;
```

MAPPING X-Y COORDINATES TO A SUB-MATRIX

Part of the process of solving a Sudoku puzzles requires the ability to identify a sub-matrix, or block, for any XY coordinate. For example, the XY coordinate (4,7), that is, row 4 and column 7, maps to sub-matrix 6. To facilitate this process, the XY coordinate values are represented as two digit numbers, as follows: 11 - 19, 21 – 29, …, 91-99 and the block values range from 1 through 9, since there are nine 3x3 sub-matrices. The following Data step generates a control input data set used to create the input format called *blkf* that maps each XY coordinate to its respective sub-matrix. A partial listing of the control data set follows the Data step and the Format procedure.

```
data blocks;
   retain fmtname 'blkf' type 'I';
   do x = 1 to 9;
      select(x);
         when(1,2,3) block=1;
         when(4,5,6) block=4;
         when(7,8,9) block=7;
         end;
      do y = 1 to 9;
         start = put(x,1.) || put(y,1.);
         label = block;
         output;
         if mod(y,3) eq 0
            then block+1;
         end;
      end;
   keep fmtname type start label;
run;

proc format cntlin=blocks;
run;
```

```
Obs    fmtname    type    start    label

  1     blkf       I       11        1
  2     blkf       I       12        1
  3     blkf       I       13        1
  4     blkf       I       14        2
  5     blkf       I       15        2
  6     blkf       I       16        2
     :          :        :        :

 75     blkf       I       93        7
 76     blkf       I       94        8
 77     blkf       I       95        8
 78     blkf       I       96        8
 79     blkf       I       97        9
 80     blkf       I       98        9
 81     blkf       I       99        9
```

Another important facet of this SAS solution requires knowing the *starting* coordinate of a given sub-matrix. For example, the 4th sub-matrix in the Sudoku grid begins at coordinate (4,1); whereas, the 9th sub-matrix begins at coordinate (7,7). The following format *$blkxy* maps easily maps each sub-matrix to its starting XY coordinate.

```
proc format;
   value $blkxy 1 = '1,1'   2 = '1,4'   3 = '1,7'
                4 = '4,1'   5 = '4,4'   6 = '4,7'
                7 = '7,1'   8 = '7,4'   9 = '7,7';
run;
```

In summary, the *blkf* format maps an XY coordinate to its respective sub-matrix, or block, and the *$blkxy* format maps a sub-matrix to its starting XY-coordinate. The ability to impute these values becomes pivotal when solving the Sudoku puzzle with respect to the sub-matrices.

Below is a listing of a SAS data set that contains five Sudoku puzzles illustrating how each observation represents a single puzzle.

```
p
u
z x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
z y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y
l 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 5 5 5 5 5
e 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5

1 . . . . 9 . . 7 . . 1 9 7 . . . . . 8 3 . . . . . . . . . 2 1 . 6 . 5 . . . . . 4
2 2 7 . 9 . . . . . . . . . . . 6 . 2 . . 6 . . . . . 7 5 . . . 2 . . 3 . . 6 . 5 . .
3 . . . . . . . 7 . . 1 9 7 . . . . . 8 3 . . . . . . . . . . . . 6 . 5 . . . . . 4
4 . . . . . . . . . . 4 . 1 . 6 . 9 . . 7 . 3 . 9 . 8 . . 1 3 . . . 7 5 . 7 . . 5 .
5 3 1 . . . . . 9 . . . 7 . . . . . 3 . 4 . 6 . . 1 . . . . 1 5 . . . . 9 4 . . 3 6

p
u
z x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
z y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y y
l 5 5 5 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9 9
e 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

1 . 7 . 3 4 . 5 3 8 . . . . 6 4 . . . 8 . . . . . . 2 . . . 4 . . . 8 . . 1 3 6 9
2 8 . . . . 2 4 . . . . . . 9 . 8 7 . . . . . . . . . 2 . . . . . 4 . . . 5 7 1 .
3 . 7 . 3 4 . . 3 8 . . . . 6 4 . . . 8 . . . . . . 2 . . . 4 . . . 8 . . 1 3 6 9
4 1 . . 8 5 . . . . . . . 6 6 . . . . . . . 1 . 5 2 . . . 8 4 . 3 . . 9 . 2 . . 5
5 2 . . 5 8 . . . . 1 6 . . . . 2 . . 3 . 8 . 5 . . . . . 4 . . . 3 . . . . . 5 1
```

INITIALIZING THE CUBE

Given a data set that contains a collection of Sudoku puzzles, we proceed to solve a puzzle. Imagine that you're looking at a transparent Sudoku grid such that you can see the cube behind the grid having likewise dimensions. Better yet, imagine a house sitting behind the square such that there must be 81 rooms with their lights on. Think about it.

Two ARRAY statements at the top of the following Data step define two data structures: the square and the cube using the *%elements* macro to define the 2 and 3-dimensional arrays.

```
 data sudoku1;
      array square{9,9} %elements(square);
      array cube{9,9,9} %elements(cube);
      set sudoku;
```

The following two DO-loops begin walking through the square, looking for non-missing values (i.e., lights on), in order to populate the cube (i.e., turn on more lights) with as much information as possible. Notice that the cube contains three types of values:

• Z-value	The value that has been assigned to the grid
• Zero	The Z-value is no longer viable
• Missing	A possible value for the grid.

For each non-missing value in the square, the XY coordinate is used to populate the cube, as follows:

- Populate the Z-vector in the cube with zeros, except for the XYZ coordinate that is assigned the value of the XY coordinate. For example, in a given puzzle, the XY coordinate (1,1) contains the value 9; thus, the cube will be assigned the values zero for the Z-vector at (1,1), except for the cell (1,1,9), which contains the value 9.

- Populate the cube with respect to the sub-matrix. Recall that a sub-matrix can manifest a digit *only once* and the cube needs to represent that vital information. Again, the coordinate (1,1) contains the value 9; thus, the Z-vectors for cells (1:3,1:3,9) will be assigned the value zero, except for the coordinate (1,1,9).

- Populate the Y-vector in the cube based on the constant values of X and Z, assigning the values 0 or Z.

- Populate the X-vector in the cube based on the constant values of Y and Z, assigning the values 0 or Z.

```
      do x = 1 to 9;
         do y = 1 to 9;
            if square{x,y} ne .
               then do;
                  xy   = put(x,1.) || put(y,1.);
                  blkx = input(scan(put(put(input(xy,blkf.),1.),$blkxy.),1),1.);
                  blky = input(scan(put(put(input(xy,blkf.),1.),$blkxy.),2),1.);
                  do z = 1 to 9;
                     if square{x,y} eq z
                        then cube{x,y,z} = z;
                        else cube{x,y,z} = 0;
                     end;
                  z = square{x,y};
                  do bx = blkx to blkx+2;
                     do by = blky to blky+2;
                        if square{bx,by} eq z
                           then cube{bx,by,z} = z;
                           else cube{bx,by,z} = 0;
                        end;
                     end;
                  do vy = 1 to 9;
                     if square{x,vy} eq z
                        then cube{x,vy,z} = z;
                        else cube{x,vy,z} = 0;
                     end;
                  do vx = 1 to 9;
                     if square{vx,y} eq z
                        then cube{vx,y,z} = z;
                        else cube{vx,y,z} = 0;
                     end;
                  end;
            end;
         end;
      keep xy:;
      drop xy;
   run;
```

Observe the following partial listings of the cube pivoting on the XY coordinate and the sub-matrix (Block 1). The first listing shows the Z-vector for the XY coordinate (1,1), which contains all zeros, except the last one, having the value 9. The second listing emphasizes a specific column (9) belonging to the Z-vector. Notice that except for the coordinate (1,1,9) the values are zero, which denotes that this value is no longer available for other cells in that sub-matrix.

```
               --------------- Z ---------------
X  Y      1  2  3      4  5  6      7  8  9
-------------------------------------------

1  1      0  0  0      0  0  0      0  0  9
   2      0  0  .      0  .  0      0  .  0
   3      0  .  .      0  0  .      0  .  0
   4      0  0  .      .  0  .      0  0  0
   5      0  0  .      0  .  .      0  0  0
   6      0  0  0      0  0  0      7  0  0
   7      0  .  .      0  0  0      0  0  0
   8      1  0  0      0  0  0      0  0  0
   9      0  0  .      0  0  0      0  .  0

2  1      0  0  0      0  0  0      0  .  0
   2      0  0  0      0  0  0      7  0  0
   3      0  0  0      4  0  0      0  0  0
   4      .  0  .      0  0  0      0  0  0
   5      0  0  .      0  0  0      0  0  0
   6      0  2  0      0  0  0      0  0  0
   7      0  0  .      0  0  0      0  0  .
   8      0  0  0      0  0  6      0  0  0
   9      0  0  0      0  5  0      0  0  0
```

```
                 --------------- Z ---------------
Block  X  Y      1  2  3      4  5  6      7  8  9
--------------------------------------------------

    1  1  1      0  0  0      0  0  0      0  0  9
          2      0  0  .      0  .  0      0  .  0
          3      0  .  .      0  0  .      0  .  0

       2  1      0  0  0      0  0  0      0  .  0
          2      0  0  0      0  0  0      7  0  0
          3      0  0  0      4  0  0      0  0  0

       3  1      0  0  0      0  .  .      0  0  0
          2      1  0  0      0  0  0      0  0  0
          3      0  .  .      0  0  .      0  0  0
```

SOLVING THE SQUARE

Once the cube has been populated either with Z-values, zeros, or missing values, the next step is to actually solve the puzzle using the method called *constraint propagation*. The following Data step processes a single observation with over 800 numeric variables including XY11-XY99, XYZ111-XYZ119, and XYZ911-XYZ999. By convention, it makes twenty attempts to solve the puzzle, and then terminates normally. But how does it solve the puzzle?

Consider an orthogonal Z-vector for a given XY coordinate on the grid. After initializing the cube, it so happens that there are *instances* where there is only *one* missing value in that vector; whereas, the others are either respective Z-values or zero. In such cases, this author claims that the missing value is the *only* possible answer for that XY coordinate.

For each iteration the program walks through the grid (cell by cell), picks up the Z-vector for that cell from the cube, determines whether there is only *one* missing value, identifies the Z-value for that missing value, and assigns it to that XY cell on the grid. Then, in the event of finding a value for the grid, the Data step proceeds to update the cube, which is similar to the initialization of the cube. Finally, it checks to determine whether there are any more empty XY cells.

Now comes a peculiar feature of the solution. It becomes necessary to perform this process pivoting on YZX and XZY coordinates, as well as the obvious XYZ coordinates. It so happens that this process requires all three positions in order to solve more difficult puzzles, thereby optimizing the cube. It is left for the reader to study that aspect of this SAS solution for solving Sudoku puzzles.

For easy Sudoku puzzles, the SAS solution solves a puzzle well within the twenty-iteration limit.=

```
data sudoku2;
     array square{9,9}  %elements(square);
     array cube{9,9,9}  %elements(cube);
     array vector{9}    v1-v9;
     set sudoku1;
     do n = 1 to 20;

      * Process X,Y,Z Coordinates ;

        do x = 1 to 9;
           do y = 1 to 9;
              do z = 1 to 9;
                 vector{z} = cube{x,y,z};
                 end;
              if nmiss(of vector{*}) eq 1
                 then do;
                    do z = 1 to 9;
                       if vector{z} eq .
                          then leave;
                       end;
                    square{x,y} = z;
                    link cube;
                    end;
              end;
           end;
```

```
         * Process Y,Z,X Coordinates ;
         * Process X,Z,Y Coordinates ;

           if nmiss(of square{*}) eq 0
              then do;
                 output;
                 stop;
                 end;
           end;
        output;
        return;

     cube:

        * Obtain the beginning X,Y coordinate of the sub-matrix ;
        * Assign Z the value of X,Y ;
        * Update the sub-matrix ;
        * Update the X-vector ;
        * Update the Y-vector ;
        return;
     run;
```

UNFAIR OR UNREASONABLE PUZZLES

An unfair Sudoku puzzle occurs when there is insufficient information to solve the puzzle without resorting to relentless trial and error. Although logical techniques help to solve Sudoku puzzles, it becomes obvious that such techniques will not accomplish the task for unfair puzzles. The trial and error approach assigns a number to a cell hardly based on logic, then making more guesses along the way. Unfortunately, when the numbers don't pan out, you must back track to where you initiated the process (i.e., guess) or, at least, to a reasonable point in order to proceed. Since the objective here is to programmatically solve Sudoku puzzles efficiently, an alternative solution is now explained.

Using the SAS solution discussed so far, the following puzzle failed to be solved. In fact, it included only several more values to the puzzle, as illustrated below.

```
          1  2  3    4  5  6    7  8  9             1  2  3    4  5  6    7  8  9
      ---------------------------------         ---------------------------------
1         5  .  .    .  .  8    2  6  .     1       5  .  .    9  .  8    2  6  3
2         .  3  .    .  .  .    1  4  7     2       9  3  8    .  .  .    1  4  7
3         .  .  .    .  .  7    9  .  .     3       .  .  .    .  .  7    9  .  .

4         .  .  .    7  9  .    .  .  .     4       .  .  .    7  9  .    .  .  1
5         6  .  .    .  8  .    .  .  4  →  5       6  .  .    .  8  .    .  .  4
6         .  .  3    .  .  .    .  9  2     6       .  .  3    .  .  .    .  9  2

7         2  8  .    4  .  .    .  1  .     7       2  8  .    4  .  .    .  1  .
8         .  .  .    .  .  .    .  .  .     8       3  .  .    .  .  .    .  .  .
9         7  1  .    .  5  .    4  .  .     9       7  1  .    .  5  .    4  .  .
```

An additional module was written to solve unfair puzzles. The solution uses a *bootstrapping* method by first identifying possible values for the empty cells and storing such information in parallel macro variables: XCOORDS, YCOORDS, and VALUES, achieved by the following several steps. Once again, the *%elements* macro generates the appropriate enumerated variables for the two data structures. Also, notice that SQL step creates the data set TRIPLETS, containing the possible values (inherent in the XYZ coordinate), ordered by the frequency of occurrence based on the X-coordinate in descending fashion. This technique increases the likelihood that the considered triplet will have the maximum affect on the cube when attempting to solve the puzzle.

```
     data xyz;
        array square{9,9}  %elements(square);
        array cube{9,9,9}  %elements(cube);
        set sudoku2;
        do x = 1 to 9;
           do y = 1 to 9;
              if square{x,y} eq .
                 then do;
                    do z = 1 to 9;
                       if cube{x,y,z} eq .
```

```
                      then output;
                   end;
                end;
          end;
       end;
    keep x y z;
 run;

 proc sql noprint;
    create table triplets as
    select count(x) as count, *
       from xyz
       group by x
       order by count desc, y, z;
 quit;

 proc sql noprint;
    select x into :xcoords separated by ' '
       from triplets;
    select y into :ycoords separated by ' '
       from triplets;
    select z into :values  separated by ' '
       from triplets;
 quit;
```

The *%bootstrap* macro uses three parameters (XCOORD, YCOORD, VALUE), the singular form of the three aforementioned macro variables, which represents a single instance of the collection of possibilities emulating the trial and error approach. The macro is much like the previous module that solves most (i.e., fair) Sudoku puzzles, except that it uses the trial and error method and proceeds as usual by updating the cube. Consequently, the puzzle gets solved, as shown below.

```
       1  2  3      4  5  6      7  8  9
----------------------------------------
1      5  4  7      9  1  8      2  6  3
2      9  3  8      5  2  6      1  4  7
3      1  6  2      3  4  7      9  5  8

4      4  2  5      7  9  3      6  8  1
5      6  9  1      2  8  5      3  7  4
6      8  7  3      1  6  4      5  9  2

7      2  8  6      4  3  9      7  1  5
8      3  5  4      6  7  1      8  2  9
9      7  1  9      8  5  2      4  3  6
```

CONCLUSION

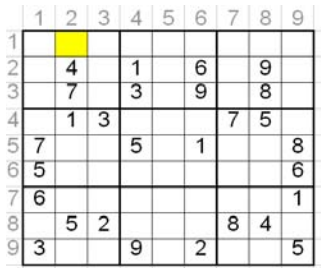
The proposed solution seems to be a natural (geometric) solution, using strictly Base SAS, in contrast to other solutions that rely heavily on SAS procedures. Also, ironically, the author has never solved a Sudoku puzzle by hand.

HOYLE – AN SQL APPROACH

This approach to solving some Sudoku puzzles was developed as an example for teaching PROC SQL as part of a graduate level class on research data management. Without a backtracking component it does not solve all possible puzzles.

This approach models three human strategies, each of which identifies a fully constrained value. That is to say each strategy fills in a cell when successful. The first strategy is to fill any empty cells that have 8 possible values eliminated.

The second strategy, sometimes called “two of three”, involves looking at the three columns that are common to a block. Within a block, if an empty cell has two values filled in the same column and the other two columns have a common value not in the block then the empty cell must have that common value. In the puzzle shown at the right, cell (1,2) must contain a 3 under this rule since there is a 3 in (9,1) and a 3 in (4,3).



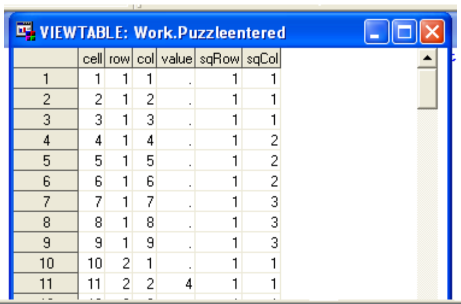
	1	2	3	4	5	6	7	8	9
1		🟨							
2		4		1		6		9	
3		7		3		9		8	
4		1	3				7	5	
5	7			5		1			8
6	5								6
7	6								1
8		5	2				8	4	
9	3			9		2			5

The third strategy applies the same logic as in strategy 2 to rows instead of columns.

These three strategies are iterated until no new cells are filled. An upper limit of 64 (81-17) iterations is also imposed since the smallest known number of initial filled values for a valid puzzle is 17.

DATA STRUCTURES

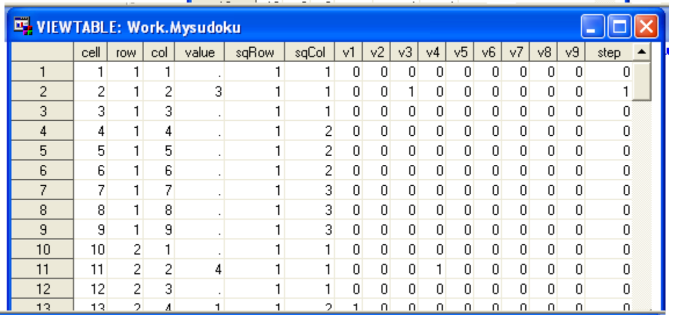
The basic data structure for this approach is to have one row for each puzzle cell with columns for the cell, row, column, row of the block (ranging from 1 to 3), and column of the block. A column for the cell value has a missing value or the filled value of each cell. The table “PuzzleEntered” at right shows the data as read in section LH1 of the code below. See, for example, that the cell in row 11 – cell (2,2) has a given value of 4 in the initial puzzle. That cell is in row 2, column 2 and its block is in the first row of blocks and the first column of blocks.



VIEWTABLE: Work.Puzzleentered

	cell	row	col	value	sqRow	sqCol
1	1	1	1	.	1	1
2	2	1	2	.	1	1
3	3	1	3	.	1	1
4	4	1	4	.	1	2
5	5	1	5	.	1	2
6	6	1	6	.	1	2
7	7	1	7	.	1	3
8	8	1	8	.	1	3
9	9	1	9	.	1	3
10	10	2	1	.	1	1
11	11	2	2	4	1	1

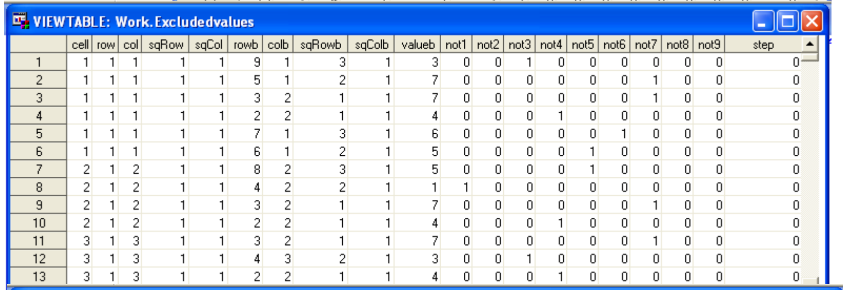
Additional Boolean columns are added to the table of known values to indicate filled values. Cell 11, with a known value of 4 shows a value of 1 for column v4 at right. This table will contain the final solution found. There is also a column indicating the last step at which the value was changed.



VIEWTABLE: Work.Mysudoku

	cell	row	col	value	sqRow	sqCol	v1	v2	v3	v4	v5	v6	v7	v8	v9	step
1	1	1	1	.	1	1	0	0	0	0	0	0	0	0	0	0
2	2	1	2	3	1	1	0	0	1	0	0	0	0	0	0	1
3	3	1	3	.	1	1	0	0	0	0	0	0	0	0	0	0
4	4	1	4	.	1	2	0	0	0	0	0	0	0	0	0	0
5	5	1	5	.	1	2	0	0	0	0	0	0	0	0	0	0
6	6	1	6	.	1	2	0	0	0	0	0	0	0	0	0	0
7	7	1	7	.	1	3	0	0	0	0	0	0	0	0	0	0
8	8	1	8	.	1	3	0	0	0	0	0	0	0	0	0	0
9	9	1	9	.	1	3	0	0	0	0	0	0	0	0	0	0
10	10	2	1	.	1	1	0	0	0	0	0	0	0	0	0	0
11	11	2	2	4	1	1	0	0	0	1	0	0	0	0	0	0
12	12	2	3	.	1	1	0	0	0	0	0	0	0	0	0	0

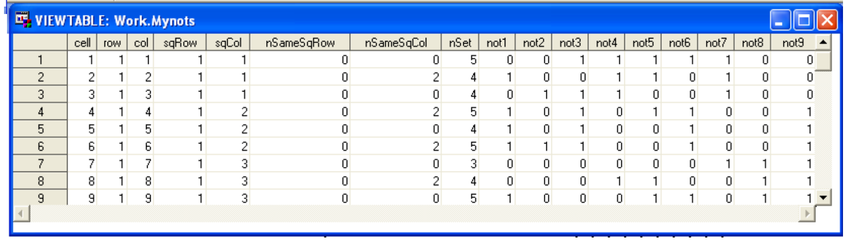
A separate table is used to track identified constraints. This table is the result of a self join (at LH6 below) matching empty cells (a) with filled cells (b). A 1 in columns not1-not9 indicates a value that cannot be used for cell (row,col).



VIEWTABLE: Work.Excludedvalues

	cell	row	col	sqRow	sqCol	rowb	colb	sqRowb	sqColb	valueb	not1	not2	not3	not4	not5	not6	not7	not8	not9	step
1	1	1	1	1	1	9	1	3	1	3	0	0	1	0	0	0	0	0	0	0
2	1	1	1	1	1	5	1	2	1	7	0	0	0	0	0	0	1	0	0	0
3	1	1	1	1	1	3	2	1	1	7	0	0	0	0	0	0	1	0	0	0
4	1	1	1	1	1	2	2	1	1	4	0	0	0	1	0	0	0	0	0	0
5	1	1	1	1	1	7	1	3	1	6	0	0	0	0	0	1	0	0	0	0
6	1	1	1	1	1	6	1	2	1	5	0	0	0	0	1	0	0	0	0	0
7	2	1	2	1	1	8	2	3	1	5	0	0	0	0	1	0	0	0	0	0
8	2	1	2	1	1	4	2	2	1	1	1	0	0	0	0	0	0	0	0	0
9	2	1	2	1	1	3	2	1	1	7	0	0	0	0	0	0	1	0	0	0
10	2	1	2	1	1	2	2	1	1	4	0	0	0	1	0	0	0	0	0	0
11	3	1	3	1	1	3	2	1	1	7	0	0	0	0	0	0	1	0	0	0
12	3	1	3	1	1	4	3	2	1	3	0	0	1	0	0	0	0	0	0	0
13	3	1	3	1	1	2	2	1	1	4	0	0	0	1	0	0	0	0	0	0

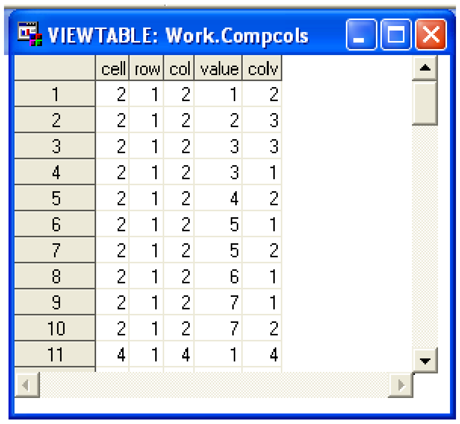
A “group by” query (at LH7 below) consolidates not1-not9 into a single table row for each puzzle cell. These columns now correspond to the third dimension in John Gerlach’s section of this paper. If the sum of these (in column nSet) is 8 then a value has been found for the cell with strategy 1. Table mustN lists any such cells (see LH9 below)



VIEWTABLE: Work.Mynots

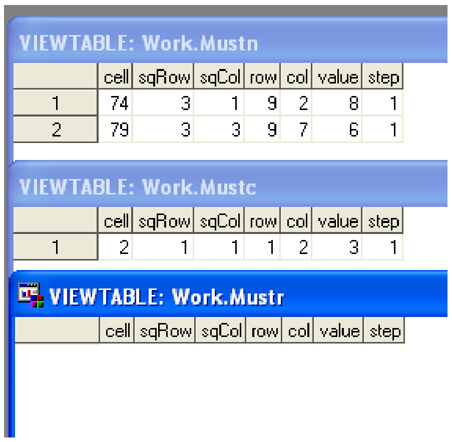
	cell	row	col	sqRow	sqCol	nSameSqRow	nSameSqCol	nSet	not1	not2	not3	not4	not5	not6	not7	not8	not9
1	1	1	1	1	1	0	0	5	0	0	1	1	1	1	1	0	0
2	2	1	2	1	1	0	2	4	1	0	0	1	1	0	1	0	0
3	3	1	3	1	1	0	0	4	0	1	1	1	0	0	1	0	0
4	4	1	4	1	2	0	2	5	1	0	1	0	1	1	0	0	1
5	5	1	5	1	2	0	0	4	1	0	1	0	0	1	0	0	1
6	6	1	6	1	2	0	2	5	1	1	1	0	0	1	0	0	1
7	7	1	7	1	3	0	0	3	0	0	0	0	0	0	1	1	1
8	8	1	8	1	3	0	2	4	0	0	0	1	1	0	0	1	1
9	9	1	9	1	3	0	0	5	1	0	0	0	1	1	0	1	1

The table compcols at right is computed to identify those cells eligible for strategy 2. For each empty cell that has two filled cells in its own column and block, it lists values that appear in the other two columns of its block, but in the other two blocks. So for the example puzzle cell (1,2) is empty and cells (2,2) and (3.2) are filled making cell (1,2) of interest. Values in rows 4-9 and columns 1 and 3 are enumerated in table Compcols. Since the value “3” appears twice for cell (1,2), strategy 2 dictates that cell (1,2) must contain a “3”. The table mustC (see LH11) lists any such cells.


VIEWTABLE: Work.Compcols

	cell	row	col	value	colv
1	2	1	2	1	2
2	2	1	2	2	3
3	2	1	2	3	3
4	2	1	2	3	1
5	2	1	2	4	2
6	2	1	2	5	1
7	2	1	2	5	2
8	2	1	2	6	1
9	2	1	2	7	1
10	2	1	2	7	2
11	4	1	4	1	4

In a similar fashion table mustR is generated for any cells found with strategy 3. The three tables – mustN, MustC, and mustR are concatenated into table mustAll (LH17) which is then used to update the solution table. (LH18 and LH 19) The figure at the right shows these three tables at the end of the first pass through the puzzle. New values have been found for cells (9.2) (9,7) and (1,2).


VIEWTABLE: Work.Mustn

	cell	sqRow	sqCol	row	col	value	step
1	74	3	1	9	2	8	1
2	79	3	3	9	7	6	1

VIEWTABLE: Work.Mustc

	cell	sqRow	sqCol	row	col	value	step
1	2	1	1	1	2	3	1

VIEWTABLE: Work.Mustr

	cell	sqRow	sqCol	row	col	value	step

REMARKS

This project implemented a small number of commonly used Sudoku strategies that find a solved value. While it could be extended to other strategies, including those which eliminate constraints, it is not clear that there exists any set of strategies which will solve any possible valid puzzle without resorting to backtracking. One use for implementing a large set of strategies could be to rate puzzles. Strategies could be ranked by difficulty according to their complexity or even by human raters. The number and difficulty of strategies needed to solve a given puzzle could be used to assign a rating to a puzzle.

```
/*   SolveSudoku.sas            */
/*    solves a Sudoku puzzle     */
/*     using SAS Proc SQL        */
/*   Larry Hoyle May 2006        */

                       *LH 1   ;

/* Enter your puzzle below      */

data PuzzleEntered;
  retain cell 0;
  do row=1 to 9;
    do col=1 to 9;
      input value 1. @@;
      cell=cell+1;
      sqRow = int((row+2)/3);
      sqCol = int((col+2)/3);
      output;
    end;
    format value row col sqRow sqCol 1.;
    format cell 2.;
    input;
  end;
datalines;
.........
.4.1.6.9.
.7.3.9.8.
.13...75.
7..5.1..8
5.......6
6.......1
.52...84.
3..9.2..5
;
run;
                             * LH 2 ;

proc sql;
  create table mySudoku as
   select *,
        case value
                when 1 then 1 else 0
                end as v1 format=1.,
         case value
                when 2 then 1 else 0
                end as v2 format=1.,
         case value
                when 3 then 1 else 0
                end as v3 format=1.,
         case value
                when 4 then 1 else 0
                end as v4 format=1.,
         case value
                when 5 then 1 else 0
                end as v5 format=1.,
         case value
                when 6 then 1 else 0
                end as v6 format=1.,
         case value
                when 7 then 1 else 0
                end as v7 format=1.,
         case value
                when 8 then 1 else 0
                end as v8 format=1.,
         case value
                when 9 then 1 else 0
                end as v9 format=1.,
                0 as step format=1.
  from PuzzleEntered;
```

```
                                * LH 3   ;
data PuzzleStatus;
  step=0;
run;

                                * LH 4   ;
%macro fillCells(mySudoku=mySudoku);
%DO %UNTIL (&upVals=0 or &step>65);

                                 * LH 5 ;
proc sql noprint;
select step into :step from PuzzleStatus;

                                 * LH 6 ;
/* join the sudoku table to itself to
   find excluded values (from b) */

 create table ExcludedValues as
   select a.cell, a.row, a.col,
          a.sqRow, a.sqCol,
          b.row as rowb,
          b.col as colb,
          b.sqRow as sqRowb,
          b.sqCol as sqColb,
          b.value as valueb,
          b.v1 as not1,
          b.v2 as not2,
          b.v3 as not3,
          b.v4 as not4,
          b.v5 as not5,
          b.v6 as not6,
          b.v7 as not7,
          b.v8 as not8,
          b.v9 as not9,
          &step as step format=1.
   from &mySudoku. as a, &mySudoku. as b
   where
                   /* empty cells (a)   */
         (a.value = .) and

             /* match filled cells (b) */
         (b.value ne .) and

                     /* not themselves */
         (a.cell ne b.cell) and
                       /* Sudoku rules */
                       /* same row */
         (a.row=b.row or
                        /* same column */
         a.col=b.col or
                        /* same square */
         (a.sqRow=b.sqRow and
          a.sqCol=b.sqCol))
   order by row,col;


                                 * LH 7 ;
/*  for the empty cells, count cells */

 create table myNots as
  select cell, row, col, sqRow, sqCol,

        /* in the same square and row, */
     sum(row=rowb and sqRow=sqRowb
         and sqCol=sqColb) as nSameSqRow
          format=1.,
      /*   in the same square and col,*/
     sum(col=colb and sqRow=sqRowb
         and sqCol=sqColb) as nSameSqCol
          format=1.,

     /*  and count and list the values
         that the cell cannot be */
     sum(max(not1),max(not2),max(not3),
         max(not4),max(not5),max(not6),
         max(not7),max(not8),max(not9))
          as nSet format=1.,

      max(not1) as not1 format=1.,
      max(not2) as not2 format=1.,
      max(not3) as not3 format=1.,
      max(not4) as not4 format=1.,
      max(not5) as not5 format=1.,
      max(not6) as not6 format=1.,
      max(not7) as not7 format=1.,
      max(not8) as not8 format=1.,
      max(not9) as not9 format=1.
  from ExcludedValues
  group by cell, row, col, sqRow, sqCol;
```

* ***LH 8*** ;

```
 update PuzzleStatus
   set step=step + 1;
 select step into :step
  from PuzzleStatus;
```

* ***STRATEGY 1 LH 9*** ;

```
 create table mustN as
   select cell, int((row+2)/3) as sqRow,
          int((col+2)/3) as sqCol,
          row, col,
          case
           when not (not1) then 1
           when not (not2) then 2
           when not (not3) then 3
           when not (not4) then 4
           when not (not5) then 5
           when not (not6) then 6
           when not (not7) then 7
           when not (not8) then 8
           when not (not9) then 9
           else . end as value,
         &step as step
      from myNots
      where nSet = 8;
```

* ***STRATEGY 2 LH 10*** ;

```
/* Within a Square, if an empty cell has
   two values filled in the same column
   and the other two columns have a
   common value not in the square
   then the empty cell must have
   that common value */

create table compCols as
 select myNots.cell,
        myNots.row,
        myNots.col,
        &mySudoku..value,
        &mySudoku..col as colv
 from myNots, &mySudoku.
 where
      /* 2 values same square and col */
   nSameSqCol=2 and
           /* filled cells in the same
              column of squares
              but not the same square */
   ((myNots.col ne &mySudoku..col and
     myNots.sqCol = &mySudoku..sqCol and
     myNots.sqRow ne &mySudoku..sqRow and
     &mySudoku..value ne .) or
      /* filled square in same column */
    (myNots.col = &mySudoku..col  and
     &mySudoku..value ne .)
    )
 order by row,col,value;
```

* ***LH 11*** ;

```
 create table mustC as
  select cell,
         int((row+2)/3) as sqRow,
         int((col+2)/3) as sqCol,
         row, col, value,
           &step as step
  from compCols
  group by row, col, value
  having count(value) = 2
              /* exclude the two cells
                 in the same square */
except
  select cell, int((row+2)/3) as sqRow,
               int((col+2)/3) as sqCol,
               row, col, value,
               &step as step
  from compCols
  where col=colv;
```

* ***STRATEGY 3 LH 12*** ;

```
      /* Same logic as Strategy 2 but
   applied to rows instead of columns */

 create table compRows as
  select myNots.cell,
         myNots.row,
         myNots.col,
         &mySudoku..value,
         &mySudoku..row as rowv
  from myNots, &mySudoku.
  where nSameSqRow=2 and
  ((myNots.row ne &mySudoku..row and
    myNots.sqRow = &mySudoku..sqRow and
    myNots.sqCol ne &mySudoku..sqCol and
     &mySudoku..value ne .) or
    (myNots.row = &mySudoku..row  and
     &mySudoku..value ne .)
   )
  order by row,col,value;
```

* ***LH 13*** ;

```
 create table mustR as
  select cell,
         int((row+2)/3) as sqRow,
         int((col+2)/3) as sqCol,
         row, col, value,
         &step as step
  from compRows
  group by row, col, value
  having count(value) = 2
except
  select cell,
         int((row+2)/3) as sqRow,
         int((col+2)/3) as sqCol,
         row, col, value,
         &step as step
  from compRows
  where row=rowv
   ;

                                            * LH 14 ;
proc sql;
  title "Cells Determined by Exclusion
Step &step ";
  select row, col, value from mustN;

                                            * LH 15 ;
  title "Cells Determined by Columns in
Square Step &step ";
  select row, col, value from mustC;

                                            * LH 16 ;
  title "Cells Determined by Rows in
Square Step &step  ";
  select row, col, value from mustR;

                                            * LH 17 ;


  create table MustAll as
   select * from mustN
    union
   select * from mustC
    union
   select * from mustR
  ;

                                            * LH 18 ;
update &mySudoku.
  set value=(select value from mustAll
             where &mySudoku..cell=
                   mustAll.cell ),
      step = &step
 where cell in(select cell from mustAll);

                                            * LH 19 ;
update &mySudoku.
  set v1 = case value
                when 1 then 1 else 0
                end ,
      v2 = case value
                when 2 then 1 else 0
                end ,
         v3 = case value
                   when 3 then 1 else 0
                   end ,
         v4 = case value
                   when 4 then 1 else 0
                   end ,
         v5 = case value
                   when 5 then 1 else 0
                   end ,
         v6 = case value
                   when 6 then 1 else 0
                   end ,
         v7 = case value
                   when 7 then 1 else 0
                   end ,
         v8 = case value
                   when 8 then 1 else 0
                   end ,
         v9 = case value
                   when 9 then 1 else 0
                   end
  ;
                                            * LH 20 ;
Title “Puzzle after Step &step”;
PROC TABULATE  DATA=&mySudoku.  ;
 VAR value;
 CLASS col /  ORDER=UNFORMATTED MISSING;
 CLASS row /  ORDER=UNFORMATTED MISSING;
 TABLE             /* Row Dimension */
Row=’ ‘,
                   /* Column Dimension */
Col=’ ‘*
   Value=’ ‘*
   Sum=’ ‘*F=1.0          ;
       ;


                                            * LH 21 ;
proc sql noprint;
 select count(row) into :upVals
   from mustALL;

%put upVals=&upVals;

%END;

%mend fillCells;

%fillCells(mysudoku=mysudoku
```

MATHEMATICAL PROGRAMMING AND SUDOKU: LINEAR OPTIMIZATION

The aim of a Sudoku puzzle is to place a group of objects into a set of positions. The discipline of Operations Research has a large subcategory devoted to such problems. The mathematical formulation begins with a set of decision variables *v{i,j}* for every possible object *i* and position *j* pairing. Each *v{i,j}* takes on the value of either 1 or 0; typically, *v{i,j}* = 1 if object *i* is placed in position *j*, and *v{i,j}* = 0 otherwise. Usually, each object can occupy at most one position, and each position can accept at most one object. If there are *N* objects and *M* positions, then these two basic conditions can be expressed as:

$$\sum_{i=1}^{N} v\{i,\, j\} \le 1 \quad \text{for each } j\text{, and} \qquad \sum_{j=1}^{M} v\{i,\, j\} \le 1 \quad \text{for each } i.$$

In the classic assignment problem of Operations Research, there is a payoff, *c{i,j}*, obtained when object *i* is placed in position *j*. Then, the total payoff can be expressed as:

$$\sum_{i=1}^{N}\sum_{j=1}^{M} c\{i,\, j\} * v\{i,\, j\}$$

and the objective is to maximize the total payoff, subject to the *(N+M)* basic conditions, which are also called constraints. The assignment problem is so common and pervasive that SAS/OR® software used to include PROC ASSIGN to solve these specific types of problems. PROC ASSIGN no longer appears in the SAS documentation for version 9, but it still can be used. The assignment problem is also a specialized form of a binary integer linear program, which can be solved in SAS with PROC LP.

Sudoku puzzles don't follow the exact set-up of the assignment problem, because they have more constraints and they don't have a payoff objective. How would one formulate a Sudoku puzzle as a binary linear integer program to feed to PROC LP? Mathematical programming problems typically have many different equivalent formulations, and the formulation can have enormous effect on the solution performance. The aim of this presentation is to find a formulation that will solve any Sudoku problem; there will not be an exploration of alternate formulations.

Since there are 81 positions and 9 digits, there are 729 variables. For the sake of discussion, number the positions consecutively from 1 – 81 going across rows and down columns, so the top row has positions 1 – 9 and the bottom row has positions 73 – 81. The variable tracking the appearance of digit 1 in position 1 will be called C01V1, the variable for digit 2 in position 1 is C01V2, the variable for digit 1 in position 2 is C02V1, etc. In the example puzzle on the first page, the initial configuration has a 4 in position 11, which becomes C11V4 = 1 mathematically, a 1 in position 13 (or C13V1 = 1), a 7 in position 20 (C20V7 = 1), a 2 in position 78 (C78V2 = 1), etc. The LP problem specification has an objective function and constraints. For Sudoku the objective is artificial, since the puzzles are constructed in order to have unique solutions; the only feasible solution is guaranteed to maximize and minimize the objective. The sum of all the variables makes an acceptable linear objective, and it must obtain the value 81, corresponding to each position being filled by exactly one digit. The variable values fixed in the initial configuration become constraints: 4 in position 11 translates into C11V4 = 1 as noted above, and that will imply that C11V1 = C11V2 = C11V3 = C11V5 = C11V6 = C11V7 = C11V8 = C11V9 = 0, etc.

The 27 conditions that define Sudoku puzzles, 9 rows, 9 columns, 9 blocks, must also be translated into specification language. These will always be the same for every puzzle, only the initial configuration changes. In order to express the fact that each digit must appear once in each row, it will be convenient to create a row constraint for each digit, such as for the first row:

C01V1+C02V1+C03V1+C04V1+C05V1+C06V1+C07V1+C08V1+C09V1 = 1
C01V2+C02V2+C03V2+C04V2+C05V2+C06V2+C07V2+C08V2+C09V2 = 1
…
C01V9+C02V9+C03V9+C04V9+C05V9+C06V9+C07V9+C08V9+C09V9 = 1

So each row has nine constraints. Similarly, each column and each block have nine such constraints; a first column constraint will look like:

C01V1+C10V1+C19V1+C28V1+C37V1+C46V1+C55V1+C64V1+C73V1 = 1

And a constraint for the upper left-hand corner block will look like:

C01V1+C02V1+C03V1+C10V1+C11V1+C12V1+C19V1+C20V1+C21V1 = 1

These 243 constraints guarantee that each row and column and block will contain exactly one of each digit; however, they don’t keep the digits from occupying the same positions. In order to do that, a constraint is needed for each position. For position 1, this will be:
C01V1+C01V2+C01V3+C01V4+C01V5+C01V6+C01V7+C01V8+C01V9 = 1
Of course, there are 81 such constraints, one for each position.

For PROC LP the objective and constraints can be packaged into an input data set; details about input data set formats can be found in the PROC LP documentation. Here is an example of how the sparse format input data set will look, with each of the type of constraints as described above (there is one additional constraint to specify that the variables are all binary):

```
  Obs       _row_       _col_    _type_    _coef_

    1    1 OBJECTIVE              MAX         .
    2    1 OBJECTIVE    C01V1                 1
    3    1 OBJECTIVE    C01V2                 1
    4    1 OBJECTIVE    C01V3                 1
    5    1 OBJECTIVE    C01V4                 1
    6    1 OBJECTIVE    C01V5                 1
    7    1 OBJECTIVE    C01V6                 1
    8    1 OBJECTIVE    C01V7                 1
    9    1 OBJECTIVE    C01V8                 1
   10    1 OBJECTIVE    C01V9                 1
   11    1 OBJECTIVE    C02V1                 1
     ...
  730    1 OBJECTIVE    C81V9                 1
  731    2 BINARY                BINARY       .
  732    2 BINARY       C01V1                 1
  733    2 BINARY       C01V2                 1
  734    2 BINARY       C01V3                 1
     ...
1460    2 BINARY         C81V9                 1
1461    3 ROW 1 VAL 1    _RHS_      EQ         1
1462    3 ROW 1 VAL 1    C01V1                 1
1463    3 ROW 1 VAL 1    C02V1                 1
1464    3 ROW 1 VAL 1    C03V1                 1
1465    3 ROW 1 VAL 1    C04V1                 1
1466    3 ROW 1 VAL 1    C05V1                 1
1467    3 ROW 1 VAL 1    C06V1                 1
1468    3 ROW 1 VAL 1    C07V1                 1
1469    3 ROW 1 VAL 1    C08V1                 1
1470    3 ROW 1 VAL 1    C09V1                 1
1471    3 ROW 1 VAL 2    _RHS_      EQ         1
1472    3 ROW 1 VAL 2    C01V2                 1
     ...
2270    3 ROW 9 VAL 9    C81V9                 1
2271    4 COL 1 VAL 1    _RHS_      EQ         1
2272    4 COL 1 VAL 1    C01V1                 1
2273    4 COL 1 VAL 1    C10V1                 1
2274    4 COL 1 VAL 1    C19V1                 1
2275    4 COL 1 VAL 1    C28V1                 1
2276    4 COL 1 VAL 1    C37V1                 1
2277    4 COL 1 VAL 1    C46V1                 1
2278    4 COL 1 VAL 1    C55V1                 1
2279    4 COL 1 VAL 1    C64V1                 1
2280    4 COL 1 VAL 1    C73V1                 1
2281    4 COL 1 VAL 2    _RHS_      EQ         1
2282    4 COL 1 VAL 2    C01V2                 1
     ...
3080    4 COL 9 VAL 9    C81V9                 1
3081    5 BOX 1 VAL 1    _RHS_      EQ         1
```

```
3082    5 BOX 1 VAL 1    C01V1                 1
3083    5 BOX 1 VAL 1    C02V1                 1
3084    5 BOX 1 VAL 1    C03V1                 1
3085    5 BOX 1 VAL 1    C10V1                 1
3086    5 BOX 1 VAL 1    C11V1                 1
3087    5 BOX 1 VAL 1    C12V1                 1
3088    5 BOX 1 VAL 1    C19V1                 1
3089    5 BOX 1 VAL 1    C20V1                 1
3090    5 BOX 1 VAL 1    C21V1                 1
3091    5 BOX 1 VAL 2    _RHS_      EQ         1
3092    5 BOX 1 VAL 2    C01V2                 1
     ...
3890    5 BOX 9 VAL 9    C81V9                 1
3891    6 CELL 01        _RHS_      EQ         1
3892    6 CELL 01        C01V1                 1
3893    6 CELL 01        C01V2                 1
3894    6 CELL 01        C01V3                 1
3895    6 CELL 01        C01V4                 1
3896    6 CELL 01        C01V5                 1
3897    6 CELL 01        C01V6                 1
3898    6 CELL 01        C01V7                 1
3899    6 CELL 01        C01V8                 1
3900    6 CELL 01        C01V9                 1
3901    6 CELL 02        _RHS_      EQ         1
3902    6 CELL 02        C02V1                 1
     ...
 4700    6 CELL 81      C81V9                 1
 4701    7 CONFIG 11    _RHS_      EQ         1
 4702    7 CONFIG 11    C11V4                 1
 4703    7 CONFIG 13    _RHS_      EQ         1
 4704    7 CONFIG 13    C13V1                 1
     ...
```

The code to produce this type of data set (starting from Hoyle's entry format), solve it with PROC LP, and then read the solution into a displayable form will be included at the end of the section.

CONSTRAINT PROGRAMMING

As noted in the previous section, while Sudoku puzzles can be expressed in the language of linear optimization, when only one possible solution exists, there is nothing to maximize or minimize. All that is required is to find a solution that satisfies the constraints. This is the domain of constraint programming, which has some of its own solution methods, distinct from linear programming. SAS/OR introduced a procedure in version 9 to solve constraint programming problems, PROC CLP. Sudoku puzzles can be expressed quite succinctly via PROC CLP.

In the formulation for PROC CLP, there will be only 81 variables, for the value at each position, which can be one of the nine digits, 1 – 9. The variables and their values can be specified in a VAR statement. The initial configuration has the form of linear constraints, which get bundled into the LINCON statement, separated by commas. The code to produce this, shown below, uses PROC SQL to read the Hoyle puzzle entry format data set and produce the initial configuration linear constraints. For the first page example puzzle, this will look like:

```
init_config_lin_con =
value11=4,value13=1,value15=6,value17=9,value20=7,value22=3,value24=9,value26=8,value29=1,
value30=3,value34=7,value35=5,value37=7,value40=5,value42=1,value45=8,value46=5,value54=6,
value55=6,value63=1,value65=5,value66=2,value70=8,value71=4,value73=3,value76=9,value78=2,
value81=5
```

The 27 Sudoku conditions are expressed easily in PROC CLP via the ALLDIFF statements; whereas the PROC LP solution used 324 constraints to express the 27 conditions, in PROC CLP one ALLDIFF statement suffices for each condition. The statement "`ALLDIFF (value1 – value9);`" requires that each of the nine variables `value1`, `value2`,…, `value9`, take on exactly one of the allowed values.

CONSTRAINTS AND SOLUTIONS

Both the PROC LP and PROC CLP solvers can find solutions even when the problem is not uniquely specified, and they will give infeasibility messages if the problem is incorrectly specified and has no solution.

It is worth noting that the 27 conditions that define Sudoku are somewhat redundant. At least five of the conditions are not necessary. Either one row or one column condition can be eliminated; to see this, note that if all nine row conditions and eight column conditions are included, then if the final column has two equal values, there will be ten of those values on the grid, which will mean that one of the rows will have two equal values, violating the row conditions. Thus, the final column condition is implied by the nine explicit row conditions and eight explicit column conditions. Once given the row and column conditions, the four "middle side" block conditions become unnecessary. To see this, note that in the first three rows, each value occurs exactly three times; if the two end block conditions are included, then each value occurs once in each of the two end blocks, leaving exactly one of each value for the middle block. This argument can be used to eliminate the middle block of the first row, the middle block of the last row, and the two end blocks of the second row, which are the middle blocks of the first and third column, respectively. However, it is not clear that eliminating the redundant conditions improves solution performance.

PROC LP SUDOKU SOLVER CODE

```
  * generate sparse data format PROC LP solver input matrix from Hoyle’s puzzle entry
  format ;
  data &ipm. (keep = _coef_  _col_   _row_   _type_) ;
   set &puzdsi. end = last ;
   LENGTH
     _row_  $13.
     _col_  $5.
     _type_  $6.
     ;

   if _N_ = 1 then do ;
   * define all the constraints except the initial configuration ;

    _coef_ = . ;
    _col_   = " " ;

    numcons + 1 ;
    _row_ = "1 OBJECTIVE" ; * Objective Function ;
    _type_ = "MAX" ;
    OUTPUT &ipm. ;

    numcons + 1 ;
    _row_ = "2 BINARY" ; * Binary variables ;
    _type_ = "BINARY" ;
    OUTPUT &ipm. ;

    _col_ = "_RHS_" ;
    _coef_ = 1 ;
    _type_ = "EQ" ;

    * row  constraints ;
    do i = 1 to 9 ;
     do j = 1 to 9 ;
      numcons + 1 ;
      _row_ = "3 ROW " || put(i,1.) || " VAL " || put(j,1.) ;
      OUTPUT &ipm. ;
     end ;
    end ;

    * column constraints ;
    do i = 1 to 9 ;
     do j = 1 to 9 ;
      numcons + 1 ;
      _row_ = "4 COL " || put(i,1.) || " VAL " || put(j,1.) ;
      OUTPUT &ipm. ;
     end ;
    end ;







    * box  constraints ;
    do i = 1 to 9 ;
     do j = 1 to 9 ;
      numcons + 1 ;
      _row_ = "5 BOX " || put(i,1.) || " VAL " || put(j,1.) ;
      OUTPUT &ipm. ;
     end ;
    end ;
    * cell constraints ;
    do i = 1 to 81 ;
      numcons + 1 ;
      _row_ = "6 CELL " || put(i,z2.) ;
      OUTPUT &ipm. ;
    end ;

   end ; * _N_ = 1 ;

   _coef_ = 1 ;
   _type_ = " " ;
   box  = (3 * (sqrow - 1)) + sqcol ;

   do j = 1 to 9 ;
    numvar + 1 ;
    _col_ = "C" || put(cell,z2.) || "V" || put(j,1.) ;
    _row_ = "1 OBJECTIVE" ;
    OUTPUT &ipm. ;
    _row_ = "2 BINARY" ;
    OUTPUT &ipm. ;
    _row_ = "3 ROW " || put(row,1.) || " VAL " || put(j,1.) ;
    OUTPUT &ipm. ;
    _row_ = "4 COL " || put(col,1.) || " VAL " || put(j,1.) ;
    OUTPUT &ipm. ;
    _row_ = "5 BOX " || put(box,1.) || " VAL " || put(j,1.) ;
    OUTPUT &ipm. ;
    _row_ = "6 CELL " || put(cell,Z2.) ;
    OUTPUT &ipm. ;
    if value = j then do ;
     numcons + 1 ;
     _row_ = "7 CONFIG " || put(cell,Z2.) ;
     OUTPUT &ipm. ;
     _col_ = "_RHS_" ;
     _type_ = "EQ" ;
     OUTPUT &ipm. ;
     _type_ = " " ;
    end ;
   end ;

   if last then do ;
    call symput("numcons",compress(put(numcons,4.))) ;
    call symput("numvar",compress(put(numvar,4.))) ;
   end ;

  run ;

  * sort constraints into logical units ;
  proc sort data = &ipm. ;
   by _row_ descending _type_ _col_ ;
  run ;


  %put       numcons      =      &numcons.    ;
  %put       numvar =      &numvar.     ;


  * feed matrix to solver, nothing special about iteration options chosen ;
  proc lp
   sparsedata
   MAXIT1=9999
   MAXIT2=9999
   imaxit=18888
   TIME=444
```

```
  data=&ipm.
  primalout=&LPSolution.
  endpause
  ;
  print best ;

 run ;

 %put _ORLP_ = ;
 %put &_ORLP_. ;

 quit / save ;

 quit ;

 * extract solution for display by PROC TABULATE ;
 data &puzdso. ;
  set &LPSolution. (where = (_type_ = "BINARY" and _value_ ge 0.9)) ;
  keep value row col ;
   i  = input(substr(_var_,2,2),2.) ;
   value = input(substr(_var_,5,1),1.) ;
   row  = ceil(i/9) ;
   col  = mod(i,9) + (9 * (mod(i,9) = 0)) ;
  * output ;
 run ;


 PROC TABULATE
  DATA = &puzdso. ;
  VAR  value ;
  CLASS col / ORDER = UNFORMATTED MISSING ;
  CLASS row / ORDER = UNFORMATTED MISSING ;
  TABLE /* Row  Dimension */
    row,
    /* Column Dimension */
    col  *
    value = "" *
    Sum  = "" * F = 1.0 ;       RUN ;
```

PROC CLP SUDOKU SOLVER CODE

```
 * generate initial configuration linear constraint macro variable ;
 proc sql ;
  reset noprint ;
  select "value" || compress(put(cell,2.)) || "=" ||
  compress(put(value,1.))
  into :init_config_lin_con separated by ","
  from &puzdsi.
  where value is not missing
  ;
 quit ;

 %put init_config_lin_con = &init_config_lin_con. ;

 proc clp out = &sudout. ;
  var  (value1 - value81) = [1,9] ;
  * initial configuration ;
  lincon &init_config_lin_con. ;
  * row  constraints ;
  alldiff (value1  - value9) ;
  alldiff (value10 - value18) ;
  alldiff (value19 - value27) ;
  alldiff (value28 - value36) ;
  alldiff (value37 - value45) ;
 alldiff (value46 - value54) ;
 alldiff (value55 - value63) ;
 alldiff (value64 - value72) ;
 alldiff (value73 - value81) ;
 * column constraints ;
 alldiff (value1 value10 value19 value28 value37 value46 value55
 value64 value73) ;
 alldiff (value2 value11 value20 value29 value38 value47 value56
 value65 value74) ;
 alldiff (value3 value12 value21 value30 value39 value48 value57
 value66 value75) ;
 alldiff (value4 value13 value22 value31 value40 value49 value58
 value67 value76) ;
 alldiff (value5 value14 value23 value32 value41 value50 value59
 value68 value77) ;
 alldiff (value6 value15 value24 value33 value42 value51 value60
 value69 value78) ;
 alldiff (value7 value16 value25 value34 value43 value52 value61
 value70 value79) ;
 alldiff (value8 value17 value26 value35 value44 value53 value62
 value71 value80) ;
 alldiff (value9 value18 value27 value36 value45 value54 value63
 value72 value81) ;
 * box  constraints ;
 alldiff (value1  - value3 value10 - value12 value19 - value21) ;
 alldiff (value4  - value6 value13 - value15 value22 - value24) ;
 alldiff (value7  - value9 value16 - value18 value25 - value27) ;
 alldiff (value28 - value30 value37 - value39 value46 - value48) ;
 alldiff (value31 - value33 value40 - value42 value49 - value51) ;
 alldiff (value34 - value36 value43 - value45 value52 - value54) ;
 alldiff (value55 - value57 value64 - value66 value73 - value75) ;
 alldiff (value58 - value60 value67 - value69 value76 - value78) ;
 alldiff (value61 - value63 value70 - value72 value79 - value81) ;
run ;


* extract solution for display by PROC TABULATE ;
data &puzdso. ;
 set &sudout. ;
 keep value row col ;
 array val{1:81} value1 - value81 ;
 do i = 1 to 81 ;
  value = val{i} ;
  row   = ceil(i/9) ;
  col   = mod(i,9) + (9 * (mod(i,9) = 0)) ;
  output ;
 end ;
 stop ;
run ;

PROC TABULATE
 DATA = &puzdso. ;
 VAR  value ;
 CLASS col / ORDER = UNFORMATTED MISSING ;
 CLASS row / ORDER = UNFORMATTED MISSING ;
 TABLE /* Row  Dimension */
   row,
   /* Column Dimension */
   col  *
   value = "" *
   Sum  = "" * F = 1.0 ;
RUN ;
```

LANGSTON

My intention in implementing a sudoku solver was to assist me with understanding what strategies were needed in solving a particular puzzle. I was not necessarily interested in the final answer to be provided by the solver, but instead how to achieve the final answer. This sudoku solver would help me to learn how to spot the proper patterns in order to solve more difficult puzzles.

Note also that this solver was not written with speed in mind. It would not likely win any shootout with other sudoku solvers written in SAS.

My implementation is based on the common strategies used to solve sudoku puzzles via candidate elimination. A "candidate" is a possible digit to appear in a given square. There are always 9 candidates to start with in any given square (unless of course the square has been assigned a particular digit by the starting puzzle). Each candidate is eliminated when the digit is the sole occupant of another square in the same row, column, or box. This elimination is the "naked single" strategy.

The "hidden single" strategy is used when a particular digit appears as a candidate in exactly one square of a row, box, or column. That digit becomes the sole occupant.

The "naked double" strategy is used when a particular pair of candidates appear as the only candidates in 2 squares of the same row/column/box. In this case, all other occurrences of these candidate digits elsewhere in the row/col/box can be eliminated.

The "hidden double" strategy is used when a particular pair of candidates appears only in 2 squares of the same row/column/box. All the other candidates appearing along with this hidden pair are eliminated in both squares.

The "naked triple" strategy is used when a particular triple of candidates appear as the only candidates in 3 squares of the same row/column/box. In this case, all other occurrences of these candidate digits elsewhere in the row/col/box can be eliminated.

There are 2 slightly different "locked candidates" strategies. Strategy 1 says that if a candidate appears only in one row of a box, then that candidate cannot appear in the same row of the other two boxes containing that row. Likewise, if a candidate appears in only one column of a box, then that candidate cannot appear in the same column of the other two boxes containing that column. Strategy 2 says that if a candidate appears only in one box for a particular row, the candidate cannot appear in the other two rows of that same box. Likewise, if a candidate appears in only one box for a particular column, the candidate cannot appear in the other 2 columns of that same box.

With the X-Wings strategy, if there is the same naked pair on two different rows and in the same columns, then any other occurrences of those digits in the same columns can be eliminated. Likewise, if there is the same naked pair on two different columns on the same rows, then any other occurrences of those digits in the same rows can be eliminated. There are further variations on the X-Wings strategy known as XY-Wings, XYZ-Wings, and Swordfish, all of which were also implemented for this solver.

My program consists of a DO WHILE(1) loop to try each of the strategies. Whenever any strategy eliminates any candidates, we CONTINUE and go back to the top of the loop, always trying the simplest strategies first.

If all strategies are attempted but with no elimination, then the program falls back on the trial-and-error method. It looks for squares with 2 possible candidates and picks one, attempting to solve based on that choice.
One of three outcomes occurs: the solution is successful, the choice is proven to be incorrect, or eventually we encounter another trial-and-error attempt. If successful, we're obviously done. If incorrect, we then know that the other candidate choice has to be the correct one so we pop the stack and proceed that way. If we encounter yet another trial-and-error attempt, we push the stack and again choose a candidate. Needless to say, the trial-and-error approach can be time-consuming, but eventually a solution will be found.

This is the basic program.

```
options missing=' ';
filename solution temp;
data _null_; file solution recfm=v lrecl=256;

     array boxindices     {9,9,2}       _temporary_;
     array whichbox       {9,9  }       _temporary_;

     array cells          {9,9,9}    $1 _temporary_;
     array origcells      {81,9,9,9} $1 _temporary_;
     array determined     {9,9  }       _temporary_;
     array origdetermined {81,9,9}      _temporary_;
     array npossibles     {9,9}         _temporary_;
     array possibles      {9,9}      $9 _temporary_;

     row=1; col=1;
     if _n_=1 then do;
        %Init_Constants;
        end;

     * this will populate the cells array with nonblank digits as
       per the puzzle description ;

     %Read_Puzzle;

     dcount=0;
     do while(1);
        if changed then have_printed=0;
        changed=0;

        array possbuff{9} $81 _temporary_;
        array ns{9,9} _temporary_;
        length digit1-digit3 $1;
        do row=1 to 9;
           do col=1 to 9;
              do digit=1 to 9;
                 substr(possibles{row,col},digit,1)=cells{row,col,digit};
                 end;
              possibles{row,col}=compress(possibles{row,col},' ');
              end;
           end;

        link perform_validation;

        if dcount=81 then do;
           put 'puzzle solved!';
           datetime = datetime();
           put 'Ending time is: ' datetime datetime25.3;
           duration = datetime - start_datetime;
           put 'Wallclock duration: ' duration time12.3;

           link print_possibles;
           leave;
           end;

        if dcount=999 then do;
           if trialstacknum>0 then do;
              link trial_value;
              continue;
              end;
           put 'PROBLEM: The original puzzle must not be valid...';
           leave;
           end;
```

```
      link naked_single;
      if changed then continue;
      link print_possibles;

      link naked_triple;
      if changed then continue;

      link hidden_single;
      if changed then continue;

      link naked_double;
      if changed then continue;

      link hidden_pairs;
      if changed then continue;

      link locked_candidates_1;
      if changed then continue;

      link locked_candidates_2;
      if changed then continue;

      link xwings_swordfish;
      if changed then continue;

      link xyzwings;
      if changed then continue;

      link xywings;
      if changed then continue;

      link trial_value;
      end;
   return;
```

The naked single strategy is implemented as follows.

```
   /*-----naked single algorithm-----*/
   do row=1 to 9;
      do col=1 to 9;
         if determined{row,col} or length(possibles{row,col})^=1 then continue;
         digit=input(possibles{row,col},1.);
         determined{row,col}=digit;
         changed=1;
         put 'Naked Single: only digit ' digit ' possible in R' row 1. 'C' col 1.;
         box=whichbox{row,col};
         do i=1 to 9;
            if i^=col and cells{row,i,digit}^=' ' then do;
               cells{row,i,digit}=' ';
               end;
            if i^=row and cells{i,col,digit}^=' ' then do;
               cells{i,col,digit}=' ';
               end;
            r=boxindices{box,i,1};
            c=boxindices{box,i,2};
            if r^=row and c^=col and cells{r,c,digit}^=' ' then do;
               cells{r,c,digit}=' ';
               end;
            end;
         end;
      end;
```

All other strategies do similar manipulations based on the description of the strategy.

The code to handle trial-and-error attempts (if all else fails) is as follows.

```
trial_value:;
        array trialrow{81} _temporary_;
        array trialcol{81} _temporary_;
        array trialnelem{81} _temporary_;
        array trialvalues{81,9} _temporary_;
        array trialelemnum{81} _temporary_;

        if dcount^=999 then do;
           trialstacknum+1;
           trialrow{trialstacknum}=.;
           trialcol{trialstacknum}=.;
           do row=1 to 9;
              do col=1 to 9;
                 do digit=1 to 9;
                    origcells{trialstacknum,row,col,digit}=cells{row,col,digit};
                    end;
                 origdetermined{trialstacknum,row,col}=determined{row,col};
                 end;
              end;
           end;
        else do while(trialstacknum>0);
           put 'PROBLEM: This value not valid...';
           trialelemnum{trialstacknum}+1;
           if trialelemnum{trialstacknum}>trialnelem{trialstacknum} then do;
              trialrow{trialstacknum} = .;
              trialstacknum=trialstacknum-1;
              end;
           else leave;
           end;
        if trialrow{trialstacknum}=. then do;
           found=0;
           do row=1 to 9 while(^found);
              do col=1 to 9 while(^found);
                 if determined{row,col} then continue;
                 do i=trialstacknum-1 to 1 by -1
                    while(^(trialrow{i}=row and trialcol{i}=col));
                    end;
                 if i^=0 then continue;
                 j=0;
                 do i=1 to 9;
                    if cells{row,col,i}=' ' then continue;
                    j+1;
                    end;
                 if j>2 then continue;
                 trialrow{trialstacknum} = row;
                 trialcol{trialstacknum} = col;
                 put 'Trial: ' row= col=;
                 j=0;
                 do i=1 to 9;
                    if cells{row,col,i}=' ' then continue;
                    j+1;
                    trialvalues{trialstacknum,j}=input(cells{row,col,i},1.);
                    put 'Trial: possible value will be ' cells{row,col,i};
                    end;
                 trialnelem{trialstacknum}=j;
                 trialelemnum{trialstacknum}=1;
                 found=1;
                 end;
              end;
           end;
```

```
         else do;
            do row=1 to 9;
               do col=1 to 9;
                  do digit=1 to 9;
                     cells{row,col,digit}=origcells{trialstacknum,row,col,digit};
                     end;
                  determined{row,col}=origdetermined{trialstacknum,row,col};
                  end;
               end;
            end;
         row = trialrow{trialstacknum};
         col = trialcol{trialstacknum};
         put row= col= trialstacknum=;
         i = trialvalues{trialstacknum,trialelemnum{trialstacknum}};
         do j=1 to 9;
            if j^=i then cells{row,col,j}=' ';
            else cells{row,col,j}=put(i,1.);
            end;
         put 'Trialing: row ' row ' col ' col ' value ' i;
         changed=1;
         return;
```

Dr. Goodnight lists sudoku among his hobbies. He found, as many fellow programmers and solvers do, that it would be interesting to develop a solver as an exercise, and of course he chose a DATA step as his foundation of implementation. Dr. Goodnight was kind enough to share his solver for the SAS Global Forum audience.

The data structures consist of _TEMPORARY_ arrays containing the various numeric values:

```
      array x[9,9]     _temporary_;     /* the puzzle matrix */
      array c[9]       _temporary_;     /* values possible for this cell */

      array sx[81,9,9] _temporary_;     /* save x matrix before guess */
      array sc[81,9]   _temporary_;     /* save c vector after guess  */

      array si[81]     _temporary_;     /* save row value of guess */
      array sj[81]     _temporary_;     /* save col value of guess */

      array cell1[9]   _temporary_ (1 1 1 4 4 4 7 7 7);  /* starting row and col of
                                                            each square */
      array cell2[9]   _temporary_ (3 3 3 6 6 6 9 9 9);  /* ending   row and col of
                                                            each square */
```

This is the outer basic loop. Each of the link routines performs a particular elimination strategy. After all routines are attempted, if the blank count is 0, we're done. Otherwise if the puzzle is still considered value, we save the blank count and make another pass. If after this pass there are no fewer cells evaluated, we go to the GUESS routine in order to start our systematic guessing process.

```
   /*- Check each cell to see if only 1 number 1-9 can possibly go there ------------*/
   /*  Check each row    to see if numbers 1-9 can go in only one cell on the row    */
   /*  Check each column to see if numbers 1-9 can go in only one cell on the column */
   /*  Check each square to see if numbers 1-9 can go in only one cell in the square */
   /*  Do this over and over until no changes were made or there are no blanks left  */

      valid=1;
      do bl=1 to 81;
         blanklast=blank;
         link single; link rowcheck; link colcheck; link sqrcheck;
         link blankcnt;
         if ^valid then do;
            Put / "The input matrix is not valid. Be sure it was entered corrrectly."/;
            goto fini;
            end;
         if blank=0 then goto fini;
         if blank=blanklast then goto guess; /* Every cell can have 2 or more values we
                                                need to guess */
         end;
   fini:
      link print;
      put / '---------------------------------------------------' /;
      goto read;
      return;
```

The 'single' link routine will determine all the possible candidates for a cell. If there turns out to be only one candidate for the cell (i.e. nmiss=1) then the cell element in x is set accordingly.

```
single:

   do i1=1 to 9;
      do j1=1 to 9;
         if x[i1,j1]=. then do; link cfill;
            if nmiss=1 then x[i1,j1]=vmiss;
            else if nmiss=0 then valid=0;
            end;
         end;
      end;

return;
```

The 'rowcheck' link routine determines which values are not yet filled into the cells of a row. If a value isn't yet placed on the row, we look at each empty cell of the row, seeing if the value already appears in its column and square. If we only find one cell where the value can go, we mark that cell. This is effectively a hidden single strategy.

```
rowcheck:

do i1=1 to 9;
   do j=1 to 9; c[j]=0; end;
   do j=1 to 9;
      v=x[i1,j];
      if v then c[v]=1;
      end;  /* set c[v]=1 if v is in row */

   do v=1 to 9;
      if c[v]=0 then do;
                    /* see if there is only 1 place in row i1 to put a v */
         nloc=0;
         do j1=1 to 9;
            if x[i1,j1]=. then do;
               found=0;
               do i=1 to 9; if x[i,j1]=v then found=1; end; /* v is in column
                                                               already */
               if found=0 then
                  do i=cell1[i1] to cell2[i1]; /* square check */
                     do j=cell1[j1] to cell2[j1];
                        if x[i,j]=v then found=1;
                        end;
                     end;
               if found=0 then do;
                  nloc+1; jloc=j1;
                  end;
               end;
            end;
         if nloc=1 then x[i1,jloc]=v;
         end;
      end;
   end;
return;
```

The 'colcheck' and 'sqrcheck' link routines do the same thing, except in column-major order and square-major order, respectively, so they are not shown here for the sake of brevity.

The 'blankcnt' link routine tells us the number of cells that still have no determined digit.

```
blankcnt:
  blank=0;
  do i4=1 to 9;
    do j4=1 to 9;
      if x[i4,j4]=. then blank+1;
      end;
    end;
  return;
```

The 'cfill' link routine is responsible for filling in the vector called c. Each element of the vector is set to 0 or 1, based on whether the digit corresponding to the element is already placed elsewhere in the same row, column, or square. Also, nmiss is set to the number of digits where the c element is 0, and vmiss is the last digit whose c element is 0. If nmiss is 1, then vmiss will be the digit to place in the cell.

```
   cfill:
      do k1=1 to 9; c[k1]=0; end;

      do k1=1 to 9;
         v=x[i1,k1];
         if v then c[v]=1;                        /* check row */
         v=x[k1,j1];
         if v then c[v]=1;                        /* check col */
         end;

      do i3=cell1[i1] to cell2[i1];
         do j3=cell1[j1] to cell2[j1];
            v=x[i3,j3];
            if v then c[v]=1;
            end;
         end;                                     /* check square */

      nmiss=0;                        /* number of values this cell can have */
      do k1=1 to 9;
         if c[k1]=0 then do;
            vmiss=k1;
            nmiss+1;
            end;
         end;
      return;
```

The heart of this solver code is the 'guess' link routine. If the simple solving techniques of 'single', 'rowcheck', 'colcheck', and 'sqrcheck' don't produce unique digits in at least one new cell for a pass, then we resort to the guessing technique. We choose the cell that has the least number of possibles, then try each one of them, repeating the process until we find that we hit an invalid configuration. We pop the stack by backing up one guess and trying a different value for the cell.

```
   guess:
      nguess=0;
      solutions=1;

   newguess:
      nguess+1;
      if nguess>81 then goto fini;

      do i=1 to 9;
         do j=1 to 9;
            sx[nguess,i,j]=x[i,j];
            end;
         end;     /* save x */
      nmin=10;
      do i1=1 to 9;        /* find the blank cell with the fewest missing values     */
         do j1=1 to 9;
            if x[i1,j1]=. then do;
               link cfill;
               if nmiss<nmin then do;
                  nmin=nmiss; imin=i1; jmin=j1;
                  end;
               end;
            end;
         end;
      if nmin=10 then goto fini;           /* cells are full */

      solutions=solutions*nmin;       /* solutions is ~ the # of possible solutions */
      si[nguess]=imin;
      sj[nguess]=jmin;
      i1=imin;
      j1=jmin;
      link cfill;
      do k=1 to 9; sc[nguess,k]=c[k]; end;                 /* save c vector */

   find:           /* setting sc[nguess,k5]=1 prevents k5 from being used again  */
                   /* if we ever fall back to this cell                          */
      do k5=1 to 9;
         if sc[nguess,k5]=0 then do;
            x[imin,jmin]=k5;
            sc[nguess,k5]=1;
            put / "Guess " nguess 2. "  X[" imin 1. "," jmin 1. "] = " k5 1. +2  /;
            link tryit;
            if blank=0 then goto fini;
            if valid then goto newguess;    /* this guess worked and all is well    */
            else link restore_x;
            end;
         end;

   /* all guesses at this level have resulted in invalid data,                */
   /* so we need to back up to previous guess cell and try another value there */

      if nguess=1 then do; put /"Failed"/; goto fini; end;
      nguess=nguess-1;
      link restore_x;
      imin=si[nguess];
      jmin=sj[nguess];
      goto find;
      return;

   restore_x:
      do i=1 to 9;
         do j=1 to 9;
            x[i,j]=sx[nguess,i,j];
            end;
         end; /* restore x */
      return;


   /*----------------------------Try current guess --------------------------------*/
   tryit:
   link blankcnt;
   link print;

   do bl=1 to 81;
      blanklast=blank;
      link single; link rowcheck; link colcheck; link sqrcheck;
```

```
      link blankcnt;
      if blank=0 then return;
      if blank=blanklast then do; /* stuck ... is current state valid */
         valid=1;
         do i1=1 to 9;
            do j1=1 to 9;
               if x[i1,j1]=. then do;
                  link cfill; /* if 1-9 already used nmiss=0 */
                  if nmiss=0 then do; valid=0; return; end;
                  end;
               end;
            end;
         return;
         end;
      end;
   return;
```

CONCLUSION

While we have not quite reached Stevens' thirteen ways of looking at this particular blackbird, we've shown a surprising number of different ways in which SAS can look at Sudoku puzzles. Nevertheless, this paper has not exhausted the possibilities for SAS based Sudoku solvers. The new PROC OPTMODEL is capable of finding Sudoku solutions and no doubt a solver could be implemented in SCL.

The data step approaches differ in the extent to which they first try to eliminate possibilities using strategies that a person might use to solve a puzzle, but in the end they all fall back on a trial and error approach to solve the most difficult puzzles. These strategies also differ in the extent to which they order their guesses, either by taking a fixed search path or by guessing in the cells with the most constraints first. DeVenezia's implementation adds a user interface which allows control of stepping through the solution process.

We showed two different SAS/OR based approaches. Both the PROC LP and CLP solutions are capable of finding multiple solutions if they exist. These could be used as part of an application to help generate new puzzles. A partial set of initial values could be entered and then values could be eliminated from one of the solutions until a puzzle with only one solution was found.

The variety of data structures used in the different approaches is interesting too, with puzzles being represented by structures ranging from a single string, to a 9X9X9 cube, to a structure with hundreds of variables.

No one approach was always faster at solving puzzles. On reflection, this might not have been unforeseen. If the cheese is on the left, the left handed mouse might be expected to find it most quickly. Dr. Goodnight's data step based solution was most frequently fastest, but for several of the puzzles tested the PROC CLP solution found a solution more quickly.

REFERENCES

Delahaye, Jean-Paul 2006. "The Science Behind Sudoku." *Scientific American* 294 (6): 80-87.

Gerlach, John. 2006. "Sudoku Puzzles - Using a Cube to Solve the Square" Atlanta: SESUG.

Heron, Andrew and Edmund James. 2005. *Sudoku for Dummies, Volume 3*. England: John Wiley & Sons, LTD,

Karwe, Vatsala, Patricia Seunarine and Carol Razafindrakoto. 2006. "Let's Sudoku with SAS®! " Philadelphia, PA: NESUG http://www.nesug.info/Proceedings/nesug06/io/io05.pdf.

Kruger, Machiel. 2006. "Using SAS to Solve Sudoku Puzzles"in "SAS Academic Program Research Newsletter Summer 2006 Edition -" http://www.sas.com/news/newsletter/academic/Research/2006_06_30.html

Matthews, Michael. 2006. "SAS, the System and Sudoku" SNUG (SAS NSW Users Group) Q1 2006 http://www.sas.com/offices/asiapacific/sp/usergroups/snug/archive/2006/presentations/Q106Matthews.pdf

SAS-L Archive; Archives of SAS-L@LISTSERV.UGA.EDU http://listserv.uga.edu/archives/sas-l.html, The University of Georgia.

Stevens, Wallace. 1997. Collected Poetry and Prose, New York: Library of America, (Kermode, F., & Richardson, J., eds.)

Wilson, Robin. 2005. *How to Solve Sudoku, A Step-by-step Guide*. England: The Infinite Ideas Company Limited,

ACKNOWLEDGMENTS

We would like to thank Dr. Goodnight for sharing his Sudoku application.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

RICHARD DEVENEZIA
Richard A. DeVenezia
9949 East Steuben Road
Remsen, NY 13438
www.devenezia.com
www.devenezia.com/contact.php

JOHN R. GERLACH
MaxisIT, Inc.
1199 Amboy Avenue,
Suite 2H,
Edison,
NJ-08837
Ph. 732-205-1717
877-MAXISIT (629-4748)
www.maxisit.com

LARRY HOYLE
Institute for Policy and Social Research
University of Kansas
1541 Lilac Road, 607 Blake
Lawrence, KS 66044-3177
785-864-9110
LarryHoyle@ku.edu
www.ipsr.ku.edu

TALBOT M. KATZ
Talbot Michael Katz
Analytic Data Information Technologies
229 East 21st Street, Apartment #2
New York, New York 10010-6459
212-460-5430
TopKatz@msn.com

RICK LANGSTON
Rick.Langston@sas.com