Sudoku Formulation as a Linear System

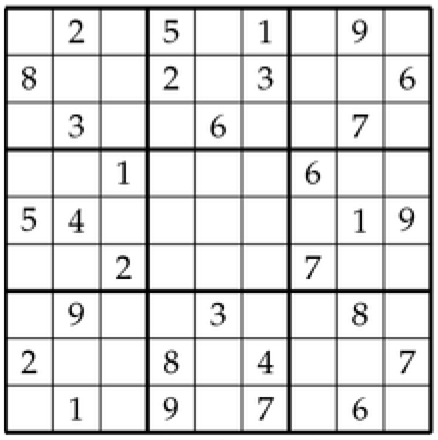
Gage Henrich

Abstract

This paper lists and describes the techniques performed to formulate 9x9 sudoku puzzles as linear programs and solve them using complex algorithms known as solvers. The sudoku ruleset was expressed as a linear system representing the constraints of an l_1-*norm* minimization to determine the "optimal" sudoku solution. LPs were formulated and solved for several sudoku puzzles using the CVX solver in Matlab.

1 Introduction

Sudoku is a logic puzzle that entails filling a 9x9 grid with digits 1 to 9 in all 9 rows, columns and 3x3 cells. Sudoku, meaning "single number" in Japanese, is generally solved by hand but can be solved using various numerical methods, ranging from backtracking algorithms to constraint programming [1]. A sample Sudoku puzzle is shown in Figure 1.



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

Figure 1: Unsolved Sudoku Puzzle

In this paper, we will explore the use of linear programming to solve a variety of Sudoku puzzles, ranging in difficuly from "medium" to "evil". We will show that a given puzzle can be formulated as an l_1-*norm* minimization with the Sudoku ruleset representing the LP constraints. Since the problem is not truly an optimization problem, the LP solution (what we are trying to minimize) may seem difficult to define.

2 LP Formulation

To formulate a Sudoku puzzle as a linear program, it is first easiest to first convert the number system from decimal to a pseudo-binary system consisting of 9-bit sequences of zeros and a single one. This will allow for us to find a solution by determining the most sparse solution (the minimum number of nonzero elements). For a vector $\boldsymbol{x}$ representing the number i, the 1 will be in the i^{th} element of $\boldsymbol{x}$. For example, the the number 7 will be represented by the sequence:

$$\boldsymbol{x}_7 = [\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ ]$$

Accordingly, we can now represent a 9x9 Sudoku matrix as a 729x1 vector, since each element in the 9x9 matrix is now represented by a 9x1 vector. This will allow us to meet all rules constraints while finding the most sparse solution. Since the optimal solution is the most sparse, it can be formulated as an l_0-norm minimization, shown in equation 1,

$$\begin{array}{rl} \text{minimize} & \|\boldsymbol{x}\|_0 \\ \text{subject to} & \boldsymbol{Ax} = \boldsymbol{b} \end{array} \tag{1}$$

where the zero-norm $\|x\|_0$ represents the number of nonzero elements of $\boldsymbol{x}$. While this accurately represents the problem, this method proves extremely difficult to formulate, and l_0-*norms* are generally hard to solve. We are essentially trying to find the sparsest solution $\boldsymbol{x}$ while meeting the Sudoku ruleset constraints. Thus, the problem can also be formulated as an l_1-*norm* minimization, which is much easier to solve. This LP is shown in equation 2,

$$\begin{array}{rl} \text{minimize} & \|\boldsymbol{x}\|_1 \\ \text{subject to} & \boldsymbol{Ax} = \boldsymbol{b} \\ & \boldsymbol{0} \leq \boldsymbol{x} \leq \boldsymbol{1} \end{array} \tag{2}$$

where:

$$\|\boldsymbol{x}\|_1 = \sum_{j=1}^{273} |\boldsymbol{x}_j| \tag{3}$$

$\boldsymbol{x}$ is the Sudoku solution of ones and zeros, and $\boldsymbol{b}$ is a vector of ones. The matrix $\boldsymbol{A}$ corresponds to the Sudoku ruleset. Its formulation will be explored in the next section.

3 Formulating the Sudoku Ruleset

For programming purposes, the ruleset of Sudoku is expressed in a total of constraints. We will formulate these constraints as their own individual matrices and compile them all to construct the matrix $\boldsymbol{A}$. When solving Sudoku by hand, there are essentially only three rules: the puzzle must contain digits 1 through 9 in all 9 rows, columns and 3x3 cells. Two additional constraints are necessary when solving the puzzle with an algorithm: the Sudoku values already given (referred to as clues) must remain in their respective positions; and every individual cell must contain a digit. The five LP constraints that make up $\boldsymbol{A}$ are thus:

- Row: Every row contains exactly one digit 1,$\cdots$,9
- Column: Every row contains exactly one digit 1,$\cdots$,9
- Cell: Every 3x3 cell contains exactly one digit 1,$\cdots$,9
- Value: Every individual sudoku cell has a digit 1,$\cdots$,9
- Clue: All clue elements must remain in their positions

The value constraint at first may seem redundant because the three prior constraints seem to already achieve it. However, it is necessary to relax the problem for the l_1-*norm* minimization [2]. The clue constraint is added so $\boldsymbol{x}$ solves the correct Sudoku problem, otherwise it would just put in arbitrary values that fit the other constraints. Combining these five matrices, the overall constraint matrix becomes:

$$\boldsymbol{A} \quad = \quad [\quad \boldsymbol{A}_{row} \quad \boldsymbol{A}_{col} \quad \boldsymbol{A}_{cell} \quad \boldsymbol{A}_{val} \quad \boldsymbol{A}_{clue} \quad]^T \tag{4}$$

We will describe the individual constraint matrices is the following subsections.

3.1 Row Constraint

We will let the matrix $\boldsymbol{A}_{row}$ denote the row constraint, which ensures that every row of the final solution contains only one digit 1,$\cdots$,9. In the decimal case, this seems like a simple formulation. However, in our case, the solution $\boldsymbol{x}$ is a 729x1 vector, since the Sudoku solution has 81 individual decimal values and each is being represented as a sequence of 9 bits. Accordingly, $\boldsymbol{A}_{row}$ needs to have 729 columns to represent all possible values. The number of rows necessary are based on the constraint: there are 9 rows, each containing nine values. Thus, $\boldsymbol{A}_{row}$ is a 81x729 matrix.

The constraint requires that the corresponding matrix has exatly one of every value for each row. For a 9x9 Sudoku puzzle the possible values 9-bit values corresponding to 1,...,9 can be expressed as a 9x9 Identity matrix.

$$\boldsymbol{I}_{9x9} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

The first row of $\boldsymbol{I}_{9x9}$ corresponds to the decimal value 1, the second row corresponds to the decimal value of 2, and so on. To ensure that every value is considered for every individual cell of every row, the $\boldsymbol{I}_{9x9}$ matrix needs to be repeated nine times for every row. Thus, we formulate the $\boldsymbol{A}_{row}$ matrix as:

$$\boldsymbol{A}_{row} = \begin{bmatrix} \boldsymbol{I}_{9x9} & \boldsymbol{I}_{9x9} & \boldsymbol{I}_{9x9} & \boldsymbol{I}_{9x9} & \boldsymbol{I}_{9x9} & \boldsymbol{I}_{9x9} & \boldsymbol{I}_{9x9} & \boldsymbol{I}_{9x9} & \boldsymbol{I}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \cdots & \mathbf{0}_{9x9} \\ \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \boldsymbol{I}_{9x9} & \boldsymbol{I}_{9x9} & \boldsymbol{I}_{9x9} & \cdots & \mathbf{0}_{9x9} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \mathbf{0}_{9x9} \\ \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \mathbf{0}_{9x9} & \cdots & \mathbf{1}_{9x9} \end{bmatrix} \tag{6}$$

It is clearly difficult to accurately and effectively portray a matrix so large in this manner. Instead, we can define a supplementary matrix, $\hat{\boldsymbol{I}}$, to make this easier to display.

$$\hat{\boldsymbol{I}} = [\ \boldsymbol{I}_{9x9}\ \ \boldsymbol{I}_{9x9}\ \ \boldsymbol{I}_{9x9}\ \ \boldsymbol{I}_{9x9}\ \ \boldsymbol{I}_{9x9}\ \ \boldsymbol{I}_{9x9}\ \ \boldsymbol{I}_{9x9}\ \ \boldsymbol{I}_{9x9}\ \ \boldsymbol{I}_{9x9}\ ] \tag{7}$$

The $\boldsymbol{A}_{row}$ matrix becomes an 81x729 diagonal matrix of $\boldsymbol{J}$. Thus, we can define the row constraint matrix as:

$$\boldsymbol{A}_{row} = \begin{bmatrix} \hat{\boldsymbol{I}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \hat{\boldsymbol{I}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \hat{\boldsymbol{I}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \hat{\boldsymbol{I}} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \hat{\boldsymbol{I}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \hat{\boldsymbol{I}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \hat{\boldsymbol{I}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hat{\boldsymbol{I}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hat{\boldsymbol{I}} \end{bmatrix} \tag{8}$$

When applied to the constraint equation $\boldsymbol{Ax} = \mathbf{1}$, the $\boldsymbol{A}_{row}$ matrix runs through all possible individual cell values and ensures that there is only one 1,$\cdots$,9 per row.

3.2 Column Constraint

The column constraint is formulated in a manner similar to that of the row constraint. Like the row constraint matrix, we know $\boldsymbol{A}_{col}$ will be 81x729 (since the Sudoku grid is square). To ensure that only one value 1,$\cdots$,9 exists in each column, every possible 9-digit string needs to be in every column of $\boldsymbol{A}_{col}$. Rather than formulating this large matrix, as we did in equation 6, we can follow the same reasoning and create another supplementary matrix to clearly define $\boldsymbol{A}_{col}$. Since there are 81 possible values for each colum, this supplementary matrix is simply an 81x81 Identity matrix, $\boldsymbol{I}_{81x81}$. Thus, to run through all nine columns, we formulate the matrix as:

$$\boldsymbol{A}_{row} = [\ \boldsymbol{I}_{81x81}\ \ \boldsymbol{I}_{81x81}\ \ \boldsymbol{I}_{81x81}\ \ \boldsymbol{I}_{81x81}\ \ \boldsymbol{I}_{81x81}\ \ \boldsymbol{I}_{81x81}\ \ \boldsymbol{I}_{81x81}\ \ \boldsymbol{I}_{81x81}\ \ \boldsymbol{I}_{81x81}\ ] \tag{9}$$

3.3 3x3 Cell Constraint

Formulation of the cell constraint matrix, $\boldsymbol{A}_{cell}$, is similar to that of the row and column constraint matrices. In formulating the cell constraint matrix, we consider again that every value needs to be present for all nine 3x3 Sudoku cells. Since each cell consists of three rows and three columns, we know we need every possible digit 1,$\cdots$,9 represented nine times for each individual cell. Since we are still dealing with 9-bit sequences, we will employ the $\boldsymbol{I}_{9x9}$ matrix, as we did in formulating the $\boldsymbol{A}_{row}$. We will introduce another supplementary matrix to make $\boldsymbol{A}_{cell}$ easier to clearly define.

$$\boldsymbol{I}^{*} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{I} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{I} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{I} \end{bmatrix} \tag{10}$$

where the $\boldsymbol{I}$ and $\boldsymbol{0}$ elements are 9x9 matrices. This matrix represents a single 3x3 cell. Rather than trying to prescribe 81 values to all nine 3x3 cells in one matrix, it is easier to account for a row of three 3x3 cells. Here, row does not a mean a single row of the Sudoku grid, but rather three rows corresponding to the height of one 3x3 cell. Thus, we introduce another supplementary matrix, $\boldsymbol{I^{**}}$ to represent a row of cells.

$$\boldsymbol{I}^{**} = \begin{bmatrix} \boldsymbol{I}^{*} & \boldsymbol{I}^{*} & \boldsymbol{I}^{*} \end{bmatrix} \tag{11}$$

Since each element is a 9x9 matrix, $\boldsymbol{I}^{**}$ is 27x729. With three 3x3 cells constructed, the final $\boldsymbol{A}_{cell}$ matrix can now be clearly defined. The final matrix requires that all nine cells have values 1,$\cdots$,9. Since $\boldsymbol{I}^{**}$ represents the a single row of cells, we need only repeat this pattern for the three columns of cells.

$$\boldsymbol{A}_{cell} = \begin{bmatrix} \boldsymbol{I}^{**} & 0 & 0 \\ 0 & \boldsymbol{I}^{**} & 0 \\ 0 & 0 & \boldsymbol{I}^{**} \end{bmatrix} \tag{12}$$

The resulting matrix has dimensions 81x729, making it the same size as the row and column constraint matrices.

3.4 Value Constraint

As it has already been stated, the $\boldsymbol{A}_{val}$ matrix is necessary to relax the l_1-*norm* minimization problem. The constraint simply requires that each individual Sudoku cell contain a value 1,$\cdots$,9. Therefore, we need to account for all possible values for each Sudoku cell. This is achieved by simply allowing each row of a 81x729 matrix represent a single cell, and placing a sequence of nine ones in the position corresponding to the cell's position. Like the previous constraint matrices, this is an extremely large matrix. However, we can create another supplementary matrix to make defining $\boldsymbol{A}_{val}$ easier. If we let:

$$\hat{\boldsymbol{I}}^{*} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{13}$$

$\boldsymbol{A}_{val}$ becomes:

$$\boldsymbol{A}_{val} = \begin{bmatrix} \hat{\boldsymbol{I}}^* & 0 & 0 & 0 & \cdots & 0 \\ 0 & \hat{\boldsymbol{I}}^* & 0 & 0 & \cdots & 0 \\ 0 & 0 & \hat{\boldsymbol{I}}^* & 0 & \cdots & 0 \\ 0 & 0 & 0 & \hat{\boldsymbol{I}}^* & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \hat{\boldsymbol{I}}^* \end{bmatrix} \tag{14}$$

3.5 Clue Constraint

The matrix corresponding to the clue constraint is the only matrix that will change with Sudoko puzzles. Accordingly, the size of the matrix will change as well, based on the number of clues given. To formulate the clue constraint matrix, it is necessary to determine the position and value of all nonzero elements in the initial puzzle matrix (clues). When these are known, we must be convert every value to 9-bit sequences. This is most easily achieved by referencing the 9x9 identity matrix for each nonzero value. Since we are only referencing the clues, number of rows of $\boldsymbol{A}_{clue}$ is equal to the number of clues in the initial Sudoku puzzle. The 729 columns represent the positions of the clues within the matrix. For example, referring to the initial Sudoku example in Figure 1 (with the the first two elements as 0 and 8, respectively), the first row of $\boldsymbol{A}_{clue}$ would be expressed in the clue matrix as:

$$\boldsymbol{A}_{clue} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 \end{bmatrix}$$

Thus, the dimensions of $\boldsymbol{A}_{clue}$ are N_cx729, where N_c equals the number of clues given. This allows for the clue constraint to be represented as a linear equality constraint and for it to be implemented in the $\boldsymbol{A}$ matrix.

3.6 Linear Equation

Combining the constraint matrices, the linear equality constraint $\boldsymbol{Ax} = \boldsymbol{b}$ becomes:

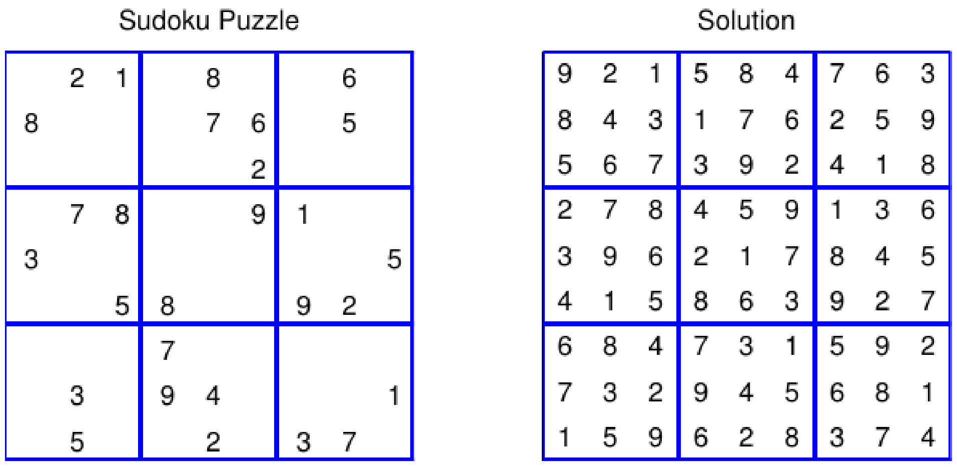
$$\begin{bmatrix} \boldsymbol{A}_{row} & \boldsymbol{A}_{col} & \boldsymbol{A}_{cell} & \boldsymbol{A}_{val} & \boldsymbol{A}_{clue} \end{bmatrix}^T \boldsymbol{x} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \tag{15}$$

4 Sudoku Solutions

The LP formulation described in the previous sections was carried out in a Matlab function. Four sample Sudoku puzzles were attempted using the solver “CVX”. The puzzles ranged in difficulty from “medium” to “evil”, as well as the “most difficult Sudoku in the world”. All code can be found in Appendix A.

4.1 Medium Difficulty

The medium level Sudoku puzzle and its solution are shown in Figure 2 below. The elapsed time to compute the solution and return the figure was 1.17 seconds.


Sudoku Puzzle
Solution

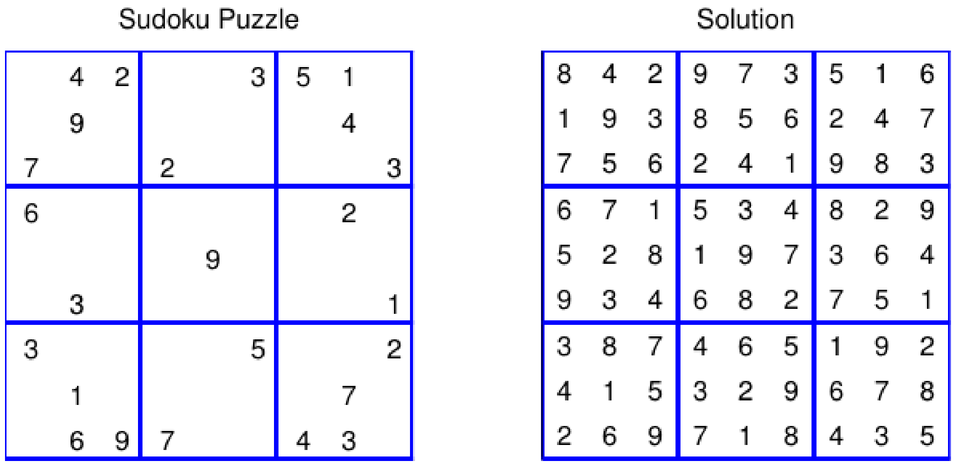
	2	1		8			6	
8				7	6		5	
					2			
	7	8			9	1		
3								5
		5	8			9	2	
			7					
	3		9	4				1
	5			2		3	7	

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

Figure 2: Medium difficulty puzzle solved in Matlab

4.2 Evil Difficulty

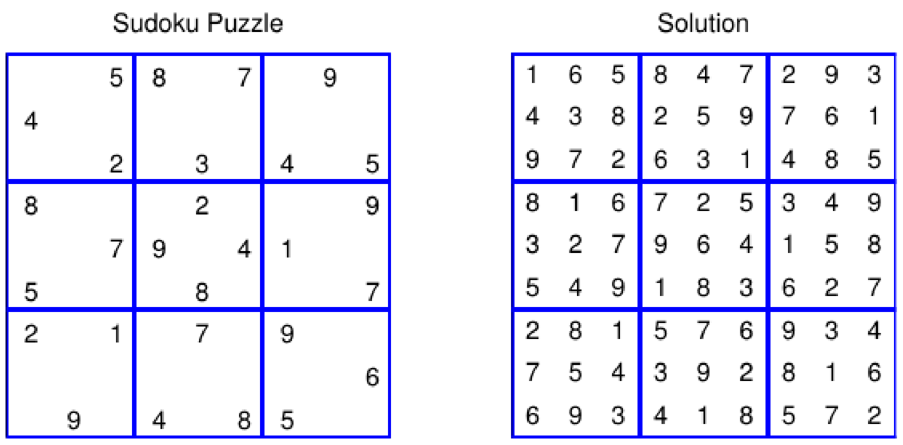
The evil level Sudoku puzzles and their solutions are shown in Figures 3 and 4. The elapsed time to compute the solution and return the first figure was 1.75 seconds. The elapsed time to compute the solution and return the second figure was 2.03 seconds.


Sudoku Puzzle
Solution

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

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

Figure 3: Evil difficulty puzzle solved in Matlab



Sudoku Puzzle

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

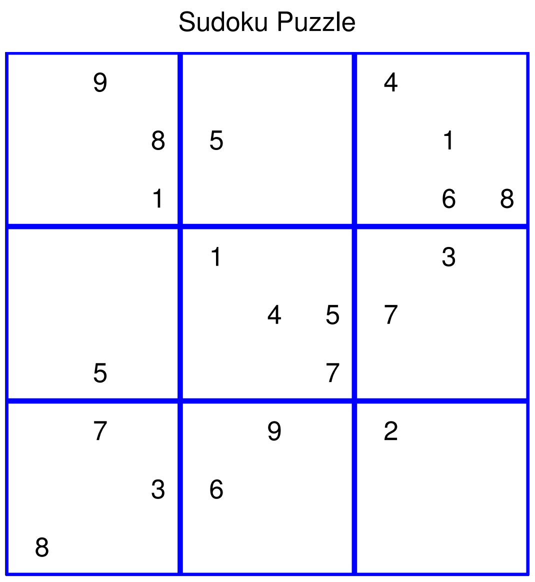
Solution

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

Figure 4: Evil difficulty puzzle solved in Matlab

4.3 Most Difficult Sudoku in the World

Unfortunately, the method here employed was unable to solve this puzzle, shown in Figure 5. While the algorithm was able to solve for some of the missing values, it was not able to complete the puzzle. It is reccomended that another solver or programming method be implemented to attempt this problem.



Sudoku Puzzle

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

Figure 5: Most difficult Sudoku in the world

A Code

```
function [soln, x] = Sudoku_Henrich(clue)

n = length(clue);
n2 = length(clue)^2;
n3 = length(clue)^3;
block = eye(n);
clues = find(clue>0);

m = size(clues);

%%% Construct matrix for 'clues' constraints
%%% Each clue element must remain in its present position for sudoku game
zer = [];
for i = 1:n
    for j = 1:n
        if clue(i,j) == 0
            zer = [zer, zeros(1,n)];
        elseif clue(i,j) > 0
            zer = [zer, block(clue(i,j),:)];
        end
    end
end
place = find(zer>0);
A_clue = zeros(m,n3);
for i = 1:m
    A_clue(i,place(i)) = zer(place(i));
end

%%% Construct matrix for column constraint
%%% Each column of sudoku matrix needs to contain all digits 1,...,9
A_col = [];
for i = 1:n
    A_col = [A_col eye(n2)];
end

%%% Create a matrix for sudoku value constraint
%%% Each individual sudoku space must contain one digit 1,...,9
place2 = [];
uno = 1;
for i = 1:n2
    place2 = [place2 uno];
    uno = uno + n;
end
```

```
A_val = zeros(n2,n3);
cells = [ones(1,n)];
for i = 1:n2
    A_val(i,place2(i):place2(i)+n-1) = cells;
end

%%% Construct matrix for row constraint
%%% Each row of sudoku matrix needs to contain all digits 1,...,9
place3 = [];
dos = 1;
I_r = [];
for i = 1:n
    I_r = [I_r eye(n)];
    place3 = [place3 dos];
    dos = dos + n2;
end
space = place2(1:n);
A_row = zeros(n2,n3);
for i = 1:n
     A_row(space(i):space(i)+n-1,place3(i):place3(i)+n2-1) = I_r;
end

%%% Construct matrix for cell constraint
%%% Each 3x3 cell of the sudoku puzzle needs to contain all digits 1,...,9
I_c1 = [];
for i = 1:sqrt(n)
    I_c1 = [I_c1 eye(n)];
end
zerz = zeros(size(I_c1));
I_c2 = [I_c1 zerz zerz];
I_c3 = [zerz I_c1 zerz];
I_c4 = [zerz zerz I_c1];
I_c5 = [I_c2; I_c3; I_c4];
J = [I_c5 I_c5 I_c5];
zerJ = zeros(size(J));
A_cell = [J zerJ zerJ; zerJ J zerJ; zerJ zerJ J];

%%% Formulate LP to solve problem with CVX
A = [A_clue; A_col; A_val; A_row; A_cell];
lenb = (4*n2 + length(clues));
b = ones(lenb,1);

%%% CVX Solver
cvx_begin
    variable x(n3)
    minimize(norm(A*x-b, 1))
    subject to
    A*x == b
    0 <= x <= 1
cvx_end

%%% Convert solution to decimal form

index = 0;
tol = 1E-3;
soln = [];
for i = 1:n
    for ii = 1:n
        for iii = 1:n
            if x(iii+index) >= 1-tol;
                soln = [soln iii];
            end
        end
        index = index + n;
    end
end
soln = reshape(soln,9,9);
soln = soln';

%%% Plot sudoku solution in figure

[yc,xc] = find(clue~=0);
c = clue(clue~=0);
sol = num2str(c(:),'%.0f');
sol = strtrim(cellstr(sol));

figure
subplot(1, 2, 1)
axis([0.25 9.25 0.25 9.25]);
axis square
for i = 1:length(c)
    vals = text(xc(i),yc(i),sol(i),'VerticalAlignment','top',...
        'HorizontalAlignment','right');
end
title('Sudoku Puzzle')
set(gca,'XTick',[])
set(gca,'YTick',[])
line([0.25 0.25],[0.25 9.25],'linewidth',2)
line([3.25 3.25],[0.25 9.25],'linewidth',2)
line([6.25 6.25],[0.25 9.25],'linewidth',2)
line([9.25 9.25],[0.25 9.25],'linewidth',2)
line([0.25 9.25],[0.25 0.25],'linewidth',2)
line([0.25 9.25],[3.25 3.25],'linewidth',2)
line([0.25 9.25],[6.25 6.25],'linewidth',2)
line([0.25 9.25],[9.25 9.25],'linewidth',2)


subplot(1, 2, 2)
axis([0 9 0 9]);
axis square
sols = num2str(soln(:),'%.0f');
sols = strtrim(cellstr(sols));
[x,y] = meshgrid(0.5:9, 0.5:9);
vals = text(x(:),y(:),sols(:),'HorizontalAlignment','center');
title('Solution')
set(gca,'XTick',[])
set(gca,'YTick',[])
line([0 0],[0 9],'linewidth',2)
line([3 3],[0 9],'linewidth',2)
line([6 6],[0 9],'linewidth',2)
line([9 9],[0 9],'linewidth',2)
line([0 9],[0 0],'linewidth',2)
line([0 9],[3 3],'linewidth',2)
line([0 9],[6 6],'linewidth',2)
line([0 9],[9 9],'linewidth',2)
```

References

[1] Wikipedia, Sudoku Solving Algorithms. *Wikipedia, The Free Encyclopedia* 2012, [Online; accessed 16-March-20013].

[2] Y. Zhang, A Simple Proof for Recoverability of L1-Minimization: Go Over or Under? Rice Univ., Houston, TX, CAAM Tech. Rep. TR05-09, 2005.