Sudoku Solver by Convex Optimization

Riley Kenyon

04/09/2020

Abstract

In this report linear and convex programming will be used to determine the solution of a Sudoku problem based on the hints given by the initial numbers. The solution does not use integer programming, but minimizes the 1-norm of the vector in order to weigh the numbering towards values of zero and one.

1 Introduction

Sudoku is a popular arithmetic based puzzle that has simple constraints. Each puzzle is a square of size N, usually 9, and is subject to the following rules

1. Each row must contain the numbers spanning 1 to N exactly once.
2. Each column must contain the numbers spanning 1 to N exactly once.
3. Each nonet must contain the numbers spanning 1 to N exactly once.

The objective is to fill in all the entries of the grid while following the rules as fast as possible. To automate the procedure and compute digitally, the problem will be posed as a linear program with constraints and a cost function. The constraints will bound the solution and incorporate the rules of play. The cost function will ultimately determine the solution that best minimizes the cost, representing the correct solution. The desired cost function here is chosen to determine the solution that contains integer numbers spanning [1,N] and satisfies the constraints. In linear programming, a method used to

solve optimization problems, the combination of constraint and criteria are written in the form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \end{array} \tag{1}$$

where the first equation represents the cost function, and the second states the problem is "subject to" the constraints. This form in particular is representative of equality constraints, however the constraints can also described by an inequality.

1.1 Problem Statement

The proposed Sudoku problem will introduce a grid of numbers with pre-populated hints that provide puzzle specific constraints for the optimization problem. The grid of numbers will be input as a matrix of size NxN and should output the Sudoku solution, as well as the solution vector x shown in equation 1. The puzzle is assumed to be solvable, and will return the original grid if no solution is found.

2 Linear Programming

To pose as a linear programming problem rather than an integer programming problem, the bounds of the solution will be relaxed and allowed to span a range from zero to one rather than forcing the solution to be purely integer. As a compensation for relaxing the variable, the cost function will be the one-norm of the vector x. As seen in figure 1, the one norm produces a weighting that favors values closest to zero. Compared to the 2-norm which is quadratic, or the infinity-norm which weighs all values equally and produces a levelized weight, the one norm produces the sparsest solution with many zeros.

For the Sudoku problem, the true values of the grid are not weighed equally if represented as the vector x in equation1. A way to manipulate the problem into a use-able form is to describe a single block within the grid as a vector that represents the possible values the block may contain. In essence, this allows the problem to describe the numbers 1 to N while maintaining uniform weighting. For example, if there are 9 possible values of the block,

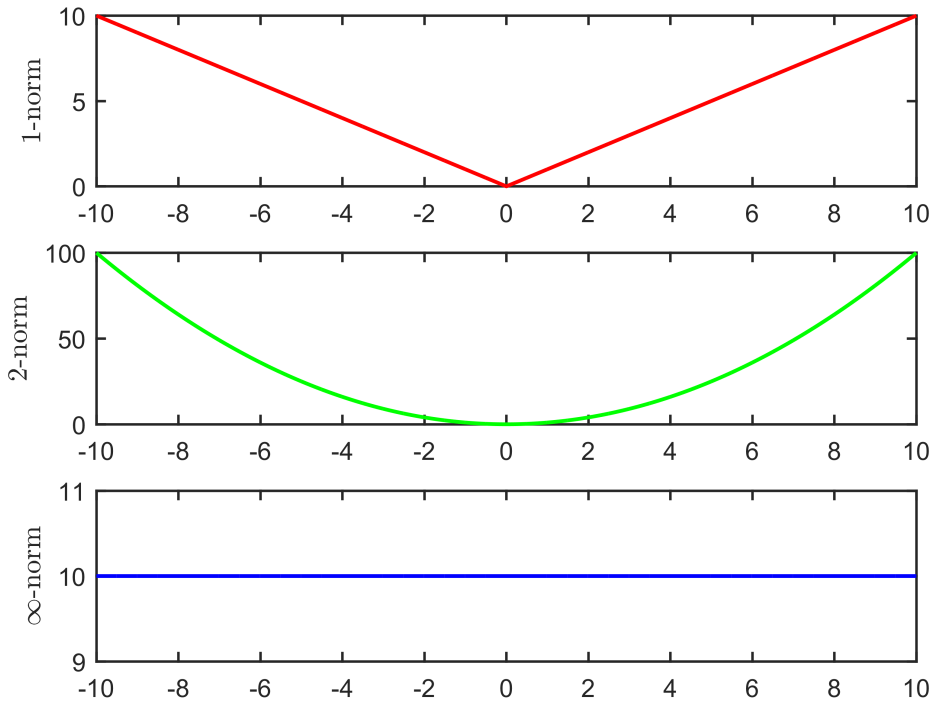

Figure 1: The weighting functions representative of the 1-norm, 2-norm, and infinity norm. Notice the 1-norm produces a weight similar in appearance to the absolute value. The weight associated with values far away from the origin is grater than the weight near zero.

the number four would be represented by the vector

$$x = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T \tag{2}$$

where the fourth entry evaluates to true. This binary approach to the problem keeps the weight across all values equivalent, i.e. the number 9 is no more favorable than 1 to the cost function. Accordingly, this expands the size of a NxN matrix to a vector of length N^3. Using this setup, the constraint matrix will incorporate the Sudoku rules.

3 Row Formulation

The row constraint manifest itself as a block within the matrix A from equation 1. The matrix represents a system of equations that describe the rule, in conjunction with the vector b that constrains the system. Mathematically, the requirement of having a single value of '1' in the first row is described by

$$x_1 + x_{N+1} + x_{2N+1} + \ldots + x_{(N-1)N+1} = b_1 \tag{3}$$

with the constraint that the number should be $b_1 = 1$ to represent only one of the possible one slots is true. As matrix multiplication, the equation has the form of

$$\begin{bmatrix} 1 & 0_{1,(N-1)} & 1 & 0_{1,(N-1)} & \ldots & 1 & 0_{1,(N-1)} \end{bmatrix} x = b_1 \tag{4}$$

where x is a vector of length N^2 and represents the available slots for the first row. The 1-norm produces the sparsest solution and ideally drives many of the values of x in equation 3 to zero. The result is ultimately persuaded to leaving one of the entities non-zero, or true in the binary sense. Extrapolating this equation to multiple slots results in a system of equations that describe the row rule for the first row of the grid, shown as

$$\begin{bmatrix} 1 & 0 & \ldots & 0 & 1 & 0 & \ldots & 0 & \ldots \\ 0 & 1 & & 0 & 0 & 1 & \ldots & 0 & \ldots \\ \vdots & & \ddots & 0 & 0 & 0 & \ddots & 0 & \ldots \\ 0 & 0 & \ldots & 1 & 0 & 0 & \ldots & 1 & \ldots \end{bmatrix} x = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} \tag{5}$$

where the second row of the matrix multiplication corresponds to $x_2 + x_{N+2} + x_{2N+2} + \ldots$, and follows the same form for subsequent rows of the matrix.

Notice that the diagonal elements are equal to one in sub-sections of size NxN. The pattern is commonly referred to as the identity matrix due to its properties to map an input to itself. The equation shown in 5 can be written as

$$A_{row,1} = \begin{bmatrix} I_N & I_N & \dots & I_N \end{bmatrix} = I_{N,N^2} \tag{6}$$

where the identity matrix of size N is repeated along the row N times. The dimension of this block is NxN^2, and encompasses the values of x_1 to x_{N^2}. This pattern holds for x encompassing the entire Sudoku grid. The starting element for the second row in the Sudoku is x_{82} assuming N is 9. This directly follows x_{81}, the last element multiplied with the first row in equation 5. Notice that by appending another row to the matrix, the Sudoku row requirement is in the form of a block diagonal matrix because the first element x_{82} directly succeeds the last element from the previous row x_{81}. By backing out the matrix multiplication, the final matrix that describes the row rule of Sudoku is described by

$$A_{row} = \begin{bmatrix} I_{1,N} & 0 & \dots & 0 \\ 0 & I_{1,N} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & I_{1,N} \end{bmatrix} \tag{7}$$

and is size N^2xN^3. Used in conjunction with matrices derived from the other Sudoku rules, the matrix A_{row} may be appended to provide the restrictions embedded from a single unique occurrence of a value per row.

4 Column Formulation

To incorporate the column rules of Sudoku into a matrix system of equations, the individual elements of x will be written out to understand the formulation. For unique elements to exist in the first column of the Sudoku grid the following equation is true

$$x_1 + x_{N^2+1} + x_{2N^2+1} + \ldots + x_{(N-1)N^2+1} = b_1 \tag{8}$$

the representation of the equation is to have elements summed within the column that are one row apart, where one row consists of N^2 elements of x. The succeeding steps are essentially equivalent to the row formulation with a distance of $N^2 - 1$ between elements

$$\begin{bmatrix} 1 & 0_{N^2-1} & 1 & 0_{N^2-1} & \dots & 1 & 0_{N^2-1} \end{bmatrix} x = b_1 \tag{9}$$

stacked on top of eachother, the format is the same as the rows, where the elements of a row are offset by one element from the previous which results in the identity matrix.The size of the identity for the column requirements is equal to N^2 whilst also only being repeated along the row. The final form of matrix that describes the column requirement is

$$A_{col} = \begin{bmatrix} I_{N^2} & I_{N^2} & \dots & I_{N^2} \end{bmatrix} \tag{10}$$

where the identity is repeated N times, for a total matrix size of $N^2 x N^3$. Used in conjunction with matrices derived from the other Sudoku rules, the matrix A_{col} may be appended to provide the restrictions embedded from a single unique occurrence of a value per column.

5 Nonet Formulation

With the typical size of a Sudoku puzzle being $9x9$, the grid can be broken into nonets to incorporate another restriction on numeric placement. The nonets each individually must have unique elements. The process is similar to preceding requirements on row and column uniqueness, however the format is different due to the strange offsets from elemnents in the flattened sense. One row below and one row suceeding is not (-1,+1), but rather +90. The case for the first nonet (top-left corner) of the Sudoku puzzle is is the equation

$$\begin{aligned} x_1 + x_{N+1} + x_{2N+1} + x_{N^2+1} + x_{N^2+N+1} + x_{N^2+2N+1} \\ + x_{2N^2+2} + x_{2N^2+N+2} + x_{2N^2+2N+2} = b_1 \end{aligned} \tag{11}$$

This can be thought about as the first three grid spaces in the first row that occupy the binary slot 1, followed by a row or separation from the first element (N^2 slots), and repeated for the third row. As a matrix, the representation of the 1 slot looks like

$$a_{blk,1} = \left[\begin{array}{cccccccc} 1 & 0_{N-1} & 1 & 0_{N-1} & 1 & 0_{N-1} & 0_{N^2-3N} & \dots \end{array} \right] \tag{12}$$

For all the available values in the first nonet, the process repeats for each element, offset by the initial position of the slot thereby creating the identity matrix. The representation for the first nonet is shown as

$$A_{blk,1,1} = \left[\begin{array}{cccccccccc} I_N & I_N & I_N & 0_{N,N^2} & \dots & I_N & I_N & I_N & 0_{NxN^2} \end{array} \right] \tag{13}$$

to incorporate the seconds block, the elements of the array are offset by $3N$ or 27 in the case of a $9x9$ Sudoku. The pattern remains the same for the remainder of the row, where the next block diagonal of the matrix occurs, the representation of the matrix is then the following for the first row of nonets

$$A_{blk,1} = \left[\begin{array}{ccccccccc} I_{N,3N} & 0 & 0 & I_{N,3N} & 0 & 0 & I_{N,3N} & 0 & 0 \\ 0 & I_{N,3N} & 0 & 0 & I_{N,3N} & 0 & 0 & I_{N,3N} & 0 \\ 0 & 0 & I_{N,3N} & 0 & 0 & I_{N,3N} & 0 & 0 & I_{N,3N} \end{array} \right] \tag{14}$$

The preceding matrix is a repeated sequence of three sub-matrices and will be used to completely encompass the matrix that represents the nonet non-repeating requirement, the block component $A_{blk,1}$ is implemented in another block diagonal matrix. The next rows of the matrix represent the second and third rows of nonets in the Sudoku puzzle.

$$A_{blk} = \left[\begin{array}{ccc} A_{blk,1} & 0 & 0 \\ 0 & A_{blk,1} & 0 \\ 0 & 0 & A_{blk,1} \end{array} \right] \tag{15}$$

where the zeros are matrices of size $Nx3N$. The requirement for the nonet rule is now incorporated into this matrix.

6 Singular Formulation

The singular requirement is embedded in the relaxed assumptions that allow X_i to span the range [0,1] rather than explicitly integers. In the binary formulation of the problem, the values of [1,9] were transformed into a vector of size 9 that used ones (true) to represent what value was present in any given square of the Sudoku. Introducing that method requires another constraint

to operate, which is that no more than one value is permitted to exist in the range of [1,9], i.e. a single grid element cannot contain multiple values. As a mathematical requirement, the representation is similar to that of rows or columns

$$A_{sing} = \begin{bmatrix} \mathbf{1}_{1xN} & 0 & \dots & 0 \\ 0 & \mathbf{1}_{1xN} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & \mathbf{1}_{1xN} \end{bmatrix} \tag{16}$$

it is a block diagonal matrix with sub-matrices of a ones vector of length $1XN$. In total, the matrix is size N^2xN^3.

7 Clue Matrices

The matrices derived from the rules of the game and the additional requirements set by the structure of the optimization problem provide many of the needed equations, however the set that ultimately makes a Sudoku problem unique is the clues. The clues of the grid are the entities that are known from the Sudoku setup. These numbers are the basis on which the optimization problem operates. The clues are converted into the binary format posed for the linear program and are appended to the end of the constraint matrix A. The decomposition from a grid from utilizes the following expression,

$$a[N(i-1) + j + G_{ij}] = 1 \tag{17}$$

Or in other words, the row vector a at the flattened position of element G_{ij} plus the value of G_{ij} is equal to 1. The flattened position is composed of $(i-1)$ multiples of length N and j is the offset to the grid position. The additional value G_{ij} converts to the correct binary position, recalled from equation 2. When multiplied by x, the respective binary value will evaluate to true. This process is continued for all elements of the Sudoku grid not equal to zero.

8 Setting up the Linear Program

With the logical requirements for the Sudoku puzzle now embedded into the system matrices, the linear program can now be defined and solved to provide

the optimal solution (i.e. the solved puzzle in binary format). Referring to the initial setup of an optimization problem in equation 1, the matrices can be described as following. The matrix A is the system of equations that define the how the rows, columns, nonets, and individual grid spaces operate with the assumption that each binary value is allowed to span the range [0,1]. To construct the matrix used with the linear program solver, the A sub-matrices will be appended to each other to incorporate all the requirements. As such, the matrix A consist of

$$A = \begin{bmatrix} A_{row} \\ A_{col} \\ A_{blk} \\ A_{sing} \\ A_{clue} \end{bmatrix} \tag{18}$$

Equally as important as the system matrix are the constraints. In this problem, the Sudoku grid is desired to have a unique element per each rule such that there are no overlap, repeating elements such that the rules are violated. This means that the constraint vector b consists solely of ones, of length m if the A matrix is size mxn. In order for the dimensions to line up for the matrix algebra, the number of constraints is equal to the number of rows present in the system matrix. The vector b used in the analysis of the problem is plainly stated as

$$b = \mathbf{1} \tag{19}$$

The optimization solver used in this analysis is the $Matlab$ routine $linprog$. The solver can utilize both equality and inequality constraints as inputs. As such, the bound of x_i to remain in the range [0,1] needs to be embedded as an inequality constraint of the form $A_{ineq}x \leq b_{ineq}$. The matrix is incorporated from the equations

$$\begin{aligned} Ix &\leq \mathbf{1} \\ Ix &\geq \mathbf{0} \end{aligned} \tag{20}$$

where the bold face 0 and 1 represent vectors of length x consisting of the value. However, due to the solver explicitly looking for a minimum, the second equation will be flipped in sign to incorporate the inequality thereby becoming $-Ix \leq \mathbf{0}$. From the equations, the values of A_{ineq} and b_{ineq} are

shown to be

$$A_{ineq} = \begin{bmatrix} I \\ -I \end{bmatrix}, b_{ineq} = \begin{bmatrix} \mathbf{1} \\ \mathbf{0} \end{bmatrix} \tag{21}$$

From the optimization formulation, there exists a cost function used to evaluate the optimal solution. For the Sudoku puzzle, because the problem was cast as a binary linear programming problem rather than an integer optimization, the cost function c will be unitary across the elements of x.

$$c = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix}^T \tag{22}$$

In essence, due to the problem constraining the elements of x to exist within the range of [0,1], then the cost function represents the 1-norm because negative entries are not allowed. As mentioned during the introduction, the 1-norm produces the sparsest solution because of the weighting shown in figure 1 which is ideal in this case due to swaying the solution to be favorable of the integer values of 0 and 1.

9 Results

The problem is fully defined and can be solved for using linear programming in *Matlab* or the auxiliary optimization libraries with *CVX*. The solution vector is obtained x, and the binary format used during the problem casting is parsed to obtain integer values and populate a solution matrix. Four example matrices used to test the setup and both *Matlab* solvers, and are described below. The time elapsed using the various methods can are shown to exhibit the efficiency of the optimization solvers. The results obtained from the example Sudoku puzzles are identical, with the only variant being the solver and time elapsed.

Solver	Puzzle 1 (sec)	Puzzle 2 (sec)	Puzzle 3 (sec)	Puzzle 4 (sec)
linprog	0.00226	0.00229	0.002139	NA
CVX	0.001632	0.001549	0.001934	NA

As seen in the table above CVX outperformed linprog by approximately 5 milliseconds, however both solvers are extremely efficient and able to solve the first three problems effectively. In the case of the fourth example, due to the way the problem was cast and because it was not posed as a true

integer optimization problem they were unable to solve. The results from the fourth puzzle included decimal values in the solution x, and could not find the integer minimum.

10 Back-Solve Technique

To solve a Sudoku incapable of being solved with the optimization proposed, the back-solve technique is implemented as a brute-force logic based combination. The brute-force method will effectively try every value that meets the rules until the full Sudoku grid is filled. The method initializes one block with the first value that meets the row, column, and nonet rules and recursively solves until it reaches a failure. When a failure occurs, all the rules cannot be satisfied, and the recursion resets or 'backs up' to the last met constraint. The process continues to iterate until complete. Although more time intensive, it is able to solve the fourth Sudoku puzzle that the optimization technique could not.

Solver	Puzzle 1 (sec)	Puzzle 2 (sec)	Puzzle 3 (sec)	Puzzle 4 (sec)
Back-Solve	0.0508	0.2490	0.0971	0.6211

Notice the time elapsed for the solver is significantly greater than the time for the optimization techniques. The comparison does not include the setup of the matrices required to solve for the linear programming approach, simply the time of the solver itself.

11 Conclusion

Each solution was determined using matlab's linear programming optimizer *linprog* to constrain each system and solve using a user-defined cost function. The 1-norm was used to create the sparsest solution and allow the solver to favor integer solutions, although the problem was not posed as an integer optimization. The option of CVX was included as an additional method to solve using the SeDuMi or SDPT3 solvers. The solutions were found to be identical to those provided using *linprog*. Both the optimization solvers were unable to solve the fourth Sudoku puzzle, however to obtain a solution the Puzzle can be solved using a 'back-solve' method based off brute force and

satisfying requirements. Using the 'back-solve' method solved the problems, but compared to the time required for the other methods the process took approximately 50-600 times as long.

A Code

A.1 Main Function

```
%% Sudoku Solver - Optimal Design
%-------------------------------------------------------------------------
% MEID: 272-513
% Riley Kenyon
% 04/09/2020
%-------------------------------------------------------------------------
clear all
close all
clc


N=9;
NN = N*N; % Number Cells

%NOTE: each cell has 9 possible numbers that can
%      be chosen.  So the first 9 variables refer
%      to cell (1,1).  If for example the (1,1) cell
%      is a 4, than x(1:9)=[0 0 0 1 0 0 0 0 0] etc
NNN = N*N*N; %Total number of binary variables x

%%%%%%%%%%%%%%%%% EXAMPLE PROBLEMS %%%%%%%%%%%%%%%%%%
% MEDIUM LEVEL
MatrixInitial1 = [0 5 0 0 2 0 3 7 0;
                  0 3 0 9 4 0 0 0 1;
                  0 0 0 7 0 0 0 0 0;
                  0 0 5 8 0 0 9 2 0;
                  3 0 0 0 0 0 0 0 5;
                  0 7 8 0 0 9 1 0 0;
                  0 0 0 0 0 2 0 0 0;
```

```
                8 0 0 0 7 6 0 5 0;
                0 2 1 0 8 0 0 6 0];

% EVIL LEVEL
MatrixInitial2 = [0 6 9 7 0 0 4 3 0;
                0 1 0 0 0 0 0 7 0;
                3 0 0 0 0 5 0 0 2;
                0 3 0 0 0 0 0 0 1;
                0 0 0 0 9 0 0 0 0;
                6 0 0 0 0 0 0 2 0;
                7 0 0 2 0 0 0 0 3;
                0 9 0 0 0 0 0 4 0;
                0 4 2 0 0 3 5 1 0];

% %EVIL LEVEL
MatrixInitial3 = [0 9 0 4 0 8 5 0 0;
                0 0 0 0 0 0 0 0 6;
                2 0 1 0 7 0 9 0 0;
                5 0 0 0 8 0 0 0 7;
                0 0 7 9 0 4 1 0 0;
                8 0 0 0 2 0 0 0 9;
                0 0 2 0 3 0 4 0 5;
                4 0 0 0 0 0 0 0 0;
                0 0 5 8 0 7 0 9 0]

% %Hardest Sudoku Ever
MatrixInitial4 = [8 0 0 0 0 0 0 0 0;
                0 0 3 6 0 0 0 0 0;
                0 7 0 0 9 0 2 0 0;
                0 5 0 0 0 7 0 0 0;
                0 0 0 0 4 5 7 0 0;
                0 0 0 1 0 0 0 3 0;
                0 0 1 0 0 0 0 6 8;
                0 0 8 5 0 0 0 1 0;
                0 9 0 0 0 0 4 0 0]
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
[MatrixFinal, x] = Sudoku_Kenyon(MatrixInitial4);
```

A.2 Functions

```
function[MatrixResult,x] = Sudoku_Kenyon(MatrixInput)
%% Sudoku Solver
% MCEN 5125 Optimal Design
% Riley Kenyon
% 04/06/2020
%-------------------------------------------------------------------------
% Description - Script to be run from SudokuProject.m
% Inputs:   Sudoku Matrix
% Outputs:  [Final Matrix, x]

%% Process
[MatrixResult,x] = solveSudoku(MatrixInput,'CVX'); % Use linprog / CVX / backsolve
end
%% Functions
function [MatrixResult,x] = solveSudoku(MatrixInput,method)
% Determine matrix dimension
[N,M] = size(MatrixInput);
if N ~= M
    error('Matrix is not Square.')
    return
end

% Populate constraint Matrices
[A,b,A_ineq,b_ineq,x_default] = setupSudoku(MatrixInput,N);

% Solves sudoku by method
switch method
    case 'linprog'
        c = ones(size(A,2),1);
        x = linprog(c,A_ineq,b_ineq,A,b);
    case 'CVX'
        x = solveCVX(A,b,1);
    case 'backsolve'
```

```
        x = x_default';
    otherwise
        error('No solver given.')
        return
end

% Check to see if solved
if(any(isnan(x)))           % Incorporate Backsolving here
    x = x_default';
end

% Get Matrix Output
tic
MatrixResult = invertSudoku(x,N);
toc
if MatrixResult == -1
    MatrixResult = MatrixInput;
end
end

function [A,b,A_ineq,b_ineq,x_default] = setupSudoku(MatrixInput,num)
    % Define appended matrix from input clues
    [appendedA,x_default] = parseSudoku(MatrixInput,num);

    % Define row requirements
    bigeye_row = repmat(eye(num),1,num);
    for i = 1:num
        A_rows_flat{i} = bigeye_row;
    end
    A_rows = blkdiag(A_rows_flat{:});

    % Define column requirements
    A_col = repmat(eye(num^2),1,num);

    % Define small cube requirements
    bigeye_block = repmat(eye(num),1,sqrt(num));
    for i = 1:sqrt(num)
        A_block_flat{i} = bigeye_block;
```

```
    end

    A_block_section = repmat(blkdiag(A_block_flat{:}),sqrt(num));

    for i = 1:sqrt(num)
        A_block_block{i} = A_block_section;
    end
    A_block = blkdiag(A_block_block{:});

    % Singular constraints
    for i = 1:num^2
        A_sing_flat{i} = ones(1,num);
    end
    A_sing = blkdiag(A_sing_flat{:});

    % Concatenate into large matrix A
    A = [A_rows; A_col; A_block; A_sing;appendedA];

    % Constraints b
    [m,n] = size(A);
    b = [ones(m,1)];

    % Bounds
    A_ineq = [eye(n); -eye(n)];
    [m_ineq,n_ineq] = size(A_ineq);
    b_ineq = [ones(m_ineq/2,1); zeros(m_ineq/2,1)];

end

function [x] = solveCVX(A,b,des_norm)
% Function to solve sudoku problem using cvx
n = size(A,2);
cvx_solver sedumi
cvx_begin
    variable x(n)
    minimize( norm(x, des_norm) )
    subject to
        A * x == b;
```

```
cvx_end
% x = round(x);
end

function [MatrixResult] = invertSudoku(x,N)
% Function to take vector input and return sudoku matrix
% Inputs:    x              (vector)
%            N              (size)
% Outputs:   MatrixResult   (Formatted Matrix
% ------------------------------------------------------------------------
ind = (1:length(x))';               % flattened indexes
x = round(x);
% if sum(x) ~=N^2

values = ind.*x;
mapped = reshape(values,N,N^2);     % where columns correspond to nine element vec
mapped = sum(mapped,1);             % sum elements in each column along all rows =
result = rem(mapped,N);          % Mapped 1->4 where 0 = 4
for ii = 1:length(mapped)
    if (mapped(ii) == 0)
        result(ii) = -1;         % case where element is not found
    elseif result(ii) == 0
        result(ii) = N;          % case where remainder is 0 == 4
    end
end
%values = values(values ~=0);        % Non-zero indexes
% remainder = rem(values,N);          % Mapped 1->4 where 0 = 4
% result = zeros(1,length(values));   % Allocate
% for ii = 1:length(values)
%     if remainder(ii) == 0
%         result(ii) = N;
%     else
%         result(ii) = remainder(ii);
%     end
% end

% Use to call backsolving method for those that cannot be solved fully with
% optimization
```

```
if any(result == -1)
    result(result == -1) = 0;
    newSol = (reshape(result,N,N))';
    MatrixResult = backsolve(newSol);
% if length(result) ~= N^2            % Should not evaluate with new method
%     MatrixResult = -1;
else
    MatrixResult = (reshape(result,N,N))';
end
end

function [appendedA,x_default] = parseSudoku(A,N)
% Function to parse sudoku input Matrix into a constraint matrix
A_vec = reshape(A',N^2,1);              % Convert to vector
loc = (1:length(A_vec))';               % Index vector
A_ind = loc(A_vec~=0);                  % Non-zero inputs

appendedA = zeros(length(A_ind),N^3);   % Allocate
x_default = zeros(1,N^3);
for ii = 1:length(A_ind)
    % Calculation - convert from # to array index, one entry per row
    appendedA(ii,N*(A_ind(ii)-1)+A_vec(A_ind(ii))) = 1;
    x_default(N*(A_ind(ii)-1)+A_vec(A_ind(ii))) = 1;
end

end
function [A] = backsolve(A)
[solved,A] = recursiveSolve(A);
if solved
    return
else
    A = -1;
end
end

function [row,col] = findEmptyPos(A)
[m,n] = size(A);
for ii = 1:m
```

```
    for jj = 1:n
        if (A(ii,jj) == 0)
            row = ii;
            col = jj;
            return
        end
    end
end
end

function [solved,row,col] = isSolved(A)
if ~any(A==0,'all') % exit condition
    row = [];
    col = [];
    solved = true;
else
    [row,col] = findEmptyPos(A);
    solved = false;
end
end

function [solved,A_in] = recursiveSolve(A_in)
global A
[solved,row,col] = isSolved(A_in);
if solved
    A = A_in;       % Set at final iteration
    return
end

for val = 1:size(A_in,1)
    if looksgood(A_in,row,col,val)
        A_in(row,col) = val;
        [solved,A_in] = recursiveSolve(A_in);
        if solved
            A_in = A;       % For each loop on the way out, set A_in to A
            return
        end
        A_in(row,col) = 0;         % If unable to solve
```

```
    end
end
solved = false;
end

function [isGood] = looksgood(A,row,col,val)
% Row check
if any(A(row,:)== val,'all')
    isGood = false;
    return
end

% Column check
if any(A(:,col) == val,'all')
    isGood = false;
    return
end

% Block check
N = sqrt(size(A,1));
startRow = row - mod(row-1,N); % 0,1,5
startCol = col - mod(col-1,N);
if any(A([startRow:startRow+N-1],[startCol:startCol+N-1]) == val,'all')
    isGood = false;
    return
end
isGood = true;

end
```