# Optimal Design Project 3

Cameron Sprenger

April 2024

## 1 Introduction

This project asks us to solve Sudoku in Matlab without using the *intlinprog* function because it can solve Sudoku with very little formulation and isn't too interesting. Instead, we have to use the standard *linprog* function and nudge the solution into the form of integer values. To quickly recap, in the game of Sudoku the player is given a 9x9 grid with only a few of the values filled in and they have to add integers into the empty squares that follow three simple rules; each row, column, and sub-grid must have each value from 1-9 without any repeats.

The code will be provided with 5 different clue matrices it has to solve which are sparse and contain only a few numbers in random indices. This will serve as a starting point for the code to populate the rest of the 9x9 matrix with numbers such that all three conditions are met. Since we aren't able to use *intlinprog* to get numbers that are integers from 1 and 9, we will have to use a trick to get the outputs in the form we want. One way to do this is to restrict the outputs to be binary with different binary combinations representing each value from 1 to 9. The easiest way to do this is to have an output of all 0's with one 1 in the index of the number it represents. For example, an output for a 2 is shown below and will be referred to as a "cell".

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

This means each of the squares in the 9x9 grid is represented by 9 binary numbers, for a total of 729 outputs for the entire grid, or $n^3$ where $n$ is the side length of the grid. Having matrices with 729 variables is quite large to show practically in this report, so all of the logic for the formulation will be shrunk to a 4x4 playing grid such that patterns can still be seen and can be shown in a more presentable way. All of the constraints scale as well as the formulation of the code I'll be presenting. An example is shown below is a solved 4x4 grid.

| | | | |
|---|---|---|---|
| 1 | 4 | 3 | 2 |
| 3 | 2 | 1 | 4 |
| 4 | 1 | 2 | 3 |
| 2 | 3 | 4 | 1 |

Figure 1: An example of a 4x4 Sudoku is below that meets each of the three constraints listed previously. There are 16 total squares in the grid, each represented by $n$ binary numbers for a total of

The new cell that represents a 2 for the smaller grid with potential integers from 1 to 4 is seen as:

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

## 2 Formulation

*linprog* is a Matlab function that solves linear programs in the form:

$$min \quad c^T x, \quad subject\ to \quad Ax \leq b \tag{3}$$

So, in order to solve the problem of Sudoku, the three constraints need to be turned into matrix form such that they can be plugged into *linprog* and solved. Additional constraints are also required to properly get the outputs in the form needed for the binary method. These additional constraints are:

- Elements in cells have to add up to 1
- Values have to be between 0 and 1
- There should only be one non-zero element in each cell

The following formulations are for each of the problem constraints, and as mentioned earlier, a playing grid of 4x4 will be used to show the patterns and formulation of the A matrices. The numbering convention shown below is how the constraints were numbered for both the 9x9 and the 4x4. Squares will be referred to as $\tilde{x_i}\ \ i = 1 : n^2$ while the unknowns in the cells will be referred to as $x_i\ \ i = 1 : n^3$.

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

Figure 2: Numbering of the 4x4 beginning at the top left and progressing left to right, top to bottom for a total of $n^2$ squares

The goal of the constraints is to push values in each of the cells to either be on or off. By constraining the problem from multiple directions, each of the values in the squares should be constrained to be an integer value if formulated correctly. If one of the constraints is formulated incorrectly, the values in the squares could be decimal values that meet the other correctly formulated constraints but violate the game rule that entries must be integers.

**Note:** The first three formulations are in standard form, and since I chose to only use the inequality function in *linprog*, any system of equations in the form $Ax = b$ are plugged into *linprog* as $Ax \leq b$ and $-Ax \leq -b$ which will be shown in section 2.8.

### 2.1 Row Constraint

As per the rules of Sudoku, each of the rows in the grid must have all of the integers from 1 to 9, or 1 to $n$ for the case of the 4x4. The first row of the 4x4 solved Sudoku above is comprised of squares numbered 1, 2, 3 and 4, each of which are represented by cells of $n$ length for a total of 16 variables. The formulation of the row constraints will be shown with the first row of the solved 4x4 Sudoku from 1, then the pattern can be scaled up to apply to all of the rows, and eventually to the full size 9x9 Sudoku.

| | | | |
|---|---|---|---|
| 1 | 4 | 3 | 2 |

Figure 3: First row squares of the solved 4x4 Sudoku, this is represented by squares 1 to 4.

Because each number in a row must be unique, there can only be a single 1 in the $i^{th}$ index of each cell for a given row. To formulate this in matrix form, the sum of the $1^{st}$ cell indices for squares 1 to 4 much add to 1. Because of the numbering system mentioned earlier, the first indices of cells 1 to 4 are represented by $x_1,\, x_5,\, x_9,\, and\, x_{13}$.

An example of the unique constraint is given below for the first row shown in equation 4.

$$\begin{aligned} \tilde{x}_1 = &\quad 1\;0\;0\;0 \\ \tilde{x}_2 = &\quad 0\;0\;0\;1 \\ \tilde{x}_3 = &\quad 0\;0\;1\;0 \\ \tilde{x}_4 = &\quad \underline{+0\;1\;0\;0} \\ &\quad 1\;1\;1\;1 \end{aligned} \tag{4}$$

Binary cells for squares 1-4 representing figure 3. Sum of the first indices in each of the rows must equal 1 to have unique values.

$$\begin{aligned} x_1 + x_5 + x_9 + x_{13} &= 1 \\ x_2 + x_6 + x_{10} + x_{14} &= 1 \\ x_3 + x_7 + x_{11} + x_{15} &= 1 \\ x_4 + x_8 + x_{12} + x_{16} &= 1 \end{aligned} \tag{5}$$

Each of the columns in the summation in equation 4 converted to the cells that comprise the squares

$$\underbrace{\left[\begin{array}{cccc|cccc|cccc|cccc|c} 1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&\vec{0} \\ 0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&\vec{0} \\ 0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&\vec{0} \\ 0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&\vec{0} \end{array}\right]}_{A_{row1}} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{64} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{6}$$

Where the first 16 elements represent the 4 cells for the first row of squares numbered 1 to 4. $\vec{0}$ are padding zero vectors of length $n^3 - n^2$ to match the number of variables. To incorporate the second row, the same pattern of identities is used, just shifted over by $n^2$ indices because this is where the cells for the second row begin. The full $A$ matrix for the row constraint is shown below.

$$\left[\begin{array}{cccc|cccc|cccc|cccc} I(n) & I(n) & I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} \\ \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) & I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} \\ \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) & I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} \\ \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) & I(n) & I(n) \end{array}\right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{64} \end{bmatrix} = \left[\underset{16x1}{\vec{1}}\right] \tag{7}$$

## 2.2 Column Constraint

Formulating the constraints for the columns is very similar to the formulation of the rows because the entries in the columns also have to be integers 1 to $n$ with the same uniqueness constraint as equation 4, the only difference being the numbering of the squares in the summation and subsequently the indexing of the cells. The biggest difference and the most challenging part of the formulation is finding the pattern of A. The best way to do this is do write out by hand the cells that make up the uniqueness summation. The following are the steps taken to reduce the first column into matrix form such that *linprog* can take it in as a constraint.

| |
|---|
| 1 |
| 3 |
| 4 |
| 2 |

Figure 4: First column squares of the solved 4x4 Sudoku comprised of squares 1, 5, 9, and 13

$$\begin{aligned} \tilde{x}_1 &= \quad 1\ 0\ 0\ 0 \\ \tilde{x}_5 &= \quad 0\ 0\ 1\ 0 \\ \tilde{x}_9 &= \quad 0\ 0\ 0\ 1 \\ \tilde{x}_{13} &= \quad \underline{+0\ 1\ 0\ 0} \\ & \quad\ \ 1\ 1\ 1\ 1 \end{aligned} \tag{8}$$

Binary cells for squares 1, 5, 9, and 13 representing figure 4. Sum of the first indices in each of the rows must equal 1 to have unique values

$$\begin{aligned} x_1 + x_{17} + x_{33} + x_{49} &= 1 \\ x_2 + x_{18} + x_{34} + x_{50} &= 1 \\ x_3 + x_{19} + x_{35} + x_{51} &= 1 \\ x_4 + x_{20} + x_{36} + x_{52} &= 1 \end{aligned} \tag{9}$$

Each of the columns in the summation in equation 8 converted to the cells that comprise the squares

$$\left[ I(n) \ \underset{nxn}{\vec{0}} \ \underset{nxn}{\vec{0}} \ \underset{nxn}{\vec{0}} \ \middle| \ I(n) \ \underset{nxn}{\vec{0}} \ \underset{nxn}{\vec{0}} \ \underset{nxn}{\vec{0}} \ \middle| \ I(n) \ \underset{nxn}{\vec{0}} \ \underset{nxn}{\vec{0}} \ \underset{nxn}{\vec{0}} \ \middle| \ I(n) \ \underset{nxn}{\vec{0}} \ \underset{nxn}{\vec{0}} \ \underset{nxn}{\vec{0}} \right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{64} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{10}$$

Expanding equation 9 to matrix form yields the $A$ matrix above which is comprised of identities of size $n$ and padding zeros in the indices that represent squares not in column 1. The location of the identities in the matrix above align with the numbering of the squares they represent, so the identity that represents square 5 is in the $5^{th}$ super element which is made of $nxn$ cells. Each of the subsequent columns begins $n^2$ indices over from the previous column, so expanding this out to the entire 4x4 playing grid yields the following $A$ matrix.

$$\left[\begin{array}{cccc|cccc|cccc|cccc} I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} \\ \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} \\ \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} \\ \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) \end{array}\right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{64} \end{bmatrix} = \begin{bmatrix} \underset{16x1}{\vec{1}} \end{bmatrix} \tag{11}$$

Where each of the rows are the constraints for the single column in the 4x4 grid. Fortunately, a diagonal of identities can be condensed into a single larger identity matrix. The reduced version of the $A$ matrix then becomes:

$$\begin{bmatrix} I(n^2) & I(n^2) & I(n^2) & I(n^2) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{64} \end{bmatrix} = \begin{bmatrix} \underset{16x1}{\vec{1}} \end{bmatrix} \tag{12}$$

## 2.3 Block Constraint

Similar to the previous two formulations, each block must have each integer between 1 and $n$, as well as the uniqueness constraint. The only difference is the indices of the cells in the constraint equations. For the 4x4 Sudoku, blocks 1, 2, 5, and 6 have to

| | |
|---|---|
| 1 | 4 |
| 3 | 2 |

Figure 5: First block squares of the solved 4x4 Sudoku comprised of squares 1, 2, 5, and 6

Since I've shown the formulation of the uniqueness constrain twice already, I will jump to the matrix formulation and the pattern that forms the $A$ matrix. For block 1, the $A$ matrix is:

$$\left[\begin{array}{cccc|cccc|c} I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{(n^3/2)xn}{\vec{0}} \end{array}\right] \tag{13}$$

And when expanded to the rest of the blocks, the pattern below arises. In essence, the pattern is $\sqrt{n}$ identities side-by-side which represent the squares in the rows of the block. These group of identities are offset by $n - \sqrt{n}$ zeros which are the other squares in the same row but in different blocks.

$$\left[\begin{array}{cccc|cccc|cccc|cccc} I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} \\ \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} \\ \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} \\ \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) & \underset{nxn}{\vec{0}} & \underset{nxn}{\vec{0}} & I(n) & I(n) \end{array}\right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{64} \end{bmatrix} = \begin{bmatrix} \underset{n^3x1}{\vec{1}} \end{bmatrix} \tag{14}$$

## 2.4 Values Between 0-1

Because we want binary outputs, two simple constraints can be added to limit the upper bound to 1, and the lower bound to 0.

$$0 \leq x_i \leq 1 \tag{15}$$

This inequality can be split into two terms, with the inequality $0 \leq x_i$ needing to be flipped by multiplying both sides by $-1$ resulting in $-x_i \leq 0$. This is done in order to meet the form of the inequality in equation 3 which is the default form of the *linprog* function.

$$\begin{aligned} x_i &\leq 1 \\ -x_i &\leq 0 \end{aligned} \tag{16}$$

Formulating the $A$ matrices for this constraint is very straight forward and is simply an identity matrix of size $n^3$ set equal to 0 or 1. The two constraints are shown below in matrix form.

$$\begin{bmatrix} I(n^3) \\ -I(n^3) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{64} \end{bmatrix} \leq \begin{bmatrix} \underset{n^3x1}{\vec{1}} \\ \underset{n^3x1}{\vec{0}} \end{bmatrix} \tag{17}$$

Where:

$$\begin{aligned} A_{\leq 1} &= \left[ I(n^3) \right] \\ A_{\geq 0} &= \left[ -I(n^3) \right] \end{aligned} \tag{18}$$

## 2.5 Cell Constraint

The sum of the values in each of the cells needs to be equal to 1 because of the binary method being used. If the cells are constrained properly, this should be because there is only a single index with a 1, with the rest of the cell being 0's. Because of the numbering convention, the cell that aligns with square $i$ is defined by the elements $[(i-1)*n+1 : i*n]$.

For the first square, the sum of the cell elements looks like:

And the expanded matrix form is:

$$\tilde{x}_1 = x_1 + x_2 + x_3 + x_4 = 1 \tag{19}$$

$$A_{cell} = \left[\begin{array}{cccc|cccc|cccc|cccc} \underset{1xn}{\vec{1}} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} \\ \vec{0} & \underset{1xn}{\vec{1}} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} \\ & & \vdots & & & & \vdots & & & & \vdots & & & & \vdots & \\ \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \underset{1xn}{\vec{1}} & \vec{0} \\ \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \vec{0} & \underset{1xn}{\vec{1}} \end{array}\right] \tag{20}$$

## 2.6 $L_1$ Norm

The last of the constraints that the binary method introduces is that only one element in each of the cells can be "on". Fortunately, by minimizing the $L_1$ norm, many of the elements are zeroed, which in our application, should help nudge the cells to only have a single value. To quickly summarize the $L_1$ norm and its formulation in linear programming are shown below.

$$min \quad |x_1| + |x_1| + \ldots + |x_n| \tag{21}$$

Each absolute value term adds an auxiliary variable which, when minimized, minimizes $|x_i|$ from the positive and negative sides. The expanded absolute values and their accompanying auxiliary variables are shown below. Since both $x_i$ and $t_i$ are unknowns, they both need to be moved to the same side of the inequality in order to solve.

$$\begin{aligned} x_1 &\le t_1 & & x_1 - t_1 \le 0 \\ -x_1 &\le t_1 & & -x_1 - t_1 \le 0 \\ &\vdots \quad \rightarrow & & \vdots \\ x_n &\le t_n & & x_n - t_n \le 0 \\ -x_n &\le t_n & & -x_n - t_n \le 0 \end{aligned} \tag{22}$$

Converting to matrix form doubles the number of unknowns because of the new auxiliary variables. In order to account for this, additional padding zeros are added to each of the other constraints that align with the auxiliary variables. This doesn't change the system of equations of the other constraints, but it is needed to solve.

$$\begin{bmatrix} I(n^3) & -I(n^3) \\ -I(n^3) & -I(n^3) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{64} \\ t_1 \\ t_2 \\ \vdots \\ t_{64} \end{bmatrix} = \begin{bmatrix} \vec{0}_{n^3x1} \\ \vec{0}_{n^3x1} \end{bmatrix} \tag{23}$$

## 2.7 Clue Matrix

In Sudoku, you are given a grid with some of the squares filled in as starting points to build off of and solve the puzzle. The more filled in squares, the easier the puzzle because the game is more constrained compared to a more sparse grid. This is because there are more forcing constraints on the new squares that you need to fill in. In order to test the program to see if it can solve certain puzzles, it needs to be fed in some clue matrices that are partially filled in. The matrices are provided as $nxn$ matrices which need to be reshaped into vectors in the same order as the unknowns ($x_i$). The *reshape* command in Matlab allows you to change a $nxn$ matrix into a vector of length $n^2$. Once this is done, the data needs to be processed into the binary form that we are using. This requires a bit of reverse engineering and can be done by making a vector of zeros called $A_{clue}$ with width $n^3$ which is the same dimension as the number of unknowns ($x_i$), and a height which is the number of filled in squares in the clue matrix. Then, each of the values in the reshaped $n^2$ vector needs to be converted into its equivalent cell. This was done by taking the value of the $i^th$ filled in square and placing it in position $A_{clue}(i, (i-1)*n+value)$. An example clue matrix and the new clue matrix are shown below.

|   |   |   |   |
|---|---|---|---|
|   |   | 4 |   |
| 1 |   |   |   |
|   |   |   | 3 |
|   | 1 |   |   |

Figure 6: Clue matrix with 4 filled in squares as starting points to complete the puzzle with

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
\underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \{0001\} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} \\
\underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \{1000\} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} \\
\underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \{0010\} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} \\
\underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}} & \{1000\} & \underset{1x4}{\vec{0}} & \underset{1x4}{\vec{0}}
\end{array}\right] \tag{24}$$

## 2.8 $\hat{A}$, and $\hat{b}$ formulation

Combining all of the formulated matrices into a single matrix with all of the constraints is simply a matter of appending each of them together and aligning the $\hat{b}$ matrix entries with the correct rows of $\hat{A}$. The constraints will be added into the $\hat{A}$ matrix in the order they were formulated with the exception that the $L_1$ norm will be at the top to easily see the auxiliary variables.

$$\hat{A} = \begin{bmatrix} I(n^3) & -I(n^3) \\ -I(n^3) & -I(n^3) \\ A_{row} & \underset{n^3xn^3}{\vec{0}} \\ -A_{row} & \underset{n^3xn^3}{\vec{0}} \\ A_{column} & \underset{n^3xn^3}{\vec{0}} \\ -A_{column} & \underset{n^3xn^3}{\vec{0}} \\ A_{block} & \underset{n^3xn^3}{\vec{0}} \\ -A_{block} & \underset{n^3xn^3}{\vec{0}} \\ A_{\leq 1} & \underset{n^3xn^3}{\vec{0}} \\ A_{\geq 0} & \underset{n^3xn^3}{\vec{0}} \\ A_{cell} & \underset{n^3xn^3}{\vec{0}} \\ -A_{cell} & \underset{n^3xn^3}{\vec{0}} \\ A_{clue} & \underset{n^3xn^3}{\vec{0}} \\ -A_{clue} & \underset{n^3xn^3}{\vec{0}} \end{bmatrix}, \hat{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{64} \\ t_1 \\ t_2 \\ \vdots \\ t_{64} \end{bmatrix}, \hat{b} = \begin{bmatrix} \vec{0} \\ \vec{0} \\ \vec{1} \\ -\vec{1} \\ \vec{1} \\ -\vec{1} \\ \vec{1} \\ -\vec{1} \\ \vec{1} \\ \vec{0} \\ \vec{1} \\ -\vec{1} \\ \vec{1} \\ -\vec{1} \end{bmatrix} \tag{25}$$

# 3 Results

After scaling the code up to solve 9x9 problems, each of the provided Sudoku puzzles was tested to see if an answer could be found given the formulated constraints. The constraints defined above should push *linprog* to output values that are either 0 or 1, which translate to integer values in each of the squares of the playing grid. If the outputs are non-integer but meet the other constraints, the final values in the squares will be decimals. For an example 4x4, if one output is {.5, 0, 0, .5}, this cell could meet all of the row, column, and block constraints as well as the sum of the cell being equal to 1. But the $L_1$ is only so strong at pushing values to 0, and the solver converged on an answer that has two elements in the cell as non-zero. The value in the square which the cell represents would be equal to;

$$(.5) * 1 + (0) * 2 + (0) * 3 + (.5) * 4 = 2.5 \tag{26}$$

Below are the provided clue matrices numbered 1 to 5, beginning with medium difficulty puzzles, to hard, and ending with the worlds hardest Sudoku puzzle that was developed to only have a single solution. The clue matrices are shown on top, and the output from *linprog* is shown below.

Clue matrices 1,2, and 4 were able to solve given the constraints above, but matrices 3 and 5 weren't able to converge to an integer solution. In the final output, there were values that were decimals which means there was the issue outlined above where the cells were comprised of not just a single integer of 1, but two elements of .5 which when multiplied out, resulted in decimal values in the squares.

In the example above the entries in position 1 and 4 were decimals meaning *linprog* came down to two choices but wasn't able to decide which of the two values to go with. In order to solve Sudoku problems 3 and 5, I added a feedback loop into the program that looked at the initial outputs from *linprog* and reads the two choices that the function narrowed the options down to and uses a brute force method of guessing and checking the possible choices. The choices are added to a $A_{Feedback}$ matrix with two rows and 1's in the column that corresponds to the index of the cell, and the $b_{feedback}$ is either [0;1] or [1;0]. This runs until the output consists of only integers.

$$\{.5,\ 0,\ 0,\ .5\} = \textbf{choices}\{1, 4\} \tag{27}$$

try:

$$x_1 = 0,\ x_4 = 1 \tag{28}$$

if no solution, try:

$$x_1 = 1,\ x_4 = 0 \tag{29}$$

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

$\Downarrow$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | **5** | 9 | 6 | **2** | 8 | **3** | **7** | 4 |
| 3 | **3** | 2 | **9** | **4** | 5 | 6 | 8 | **1** |
| 6 | 8 | 4 | **7** | 3 | 1 | 5 | 9 | 2 |
| 4 | 1 | **5** | **8** | 6 | 3 | **9** | **2** | 7 |
| **3** | 9 | 6 | 2 | 1 | 7 | 8 | 4 | **5** |
| 2 | **7** | **8** | 4 | 5 | **9** | **1** | 3 | 6 |
| 5 | 6 | 7 | 3 | 9 | **2** | 4 | 1 | 8 |
| **8** | 4 | 3 | 1 | **7** | **6** | 2 | **5** | 9 |
| 9 | **2** | **1** | 5 | **8** | 4 | 7 | **6** | 3 |

Figure 7: Clue matrix #1 and the solved puzzle

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

$$\Downarrow$$

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| 2 | **6** | **9** | **7** | 1 | 8 | **4** | **3** | 5 |
| 4 | **1** | 5 | 3 | 2 | 9 | 6 | **7** | 8 |
| **3** | 8 | 7 | 4 | 6 | **5** | 1 | 9 | **2** |
| 9 | **3** | 4 | 6 | 8 | 2 | 7 | 5 | **1** |
| 5 | 2 | 8 | 1 | **9** | 7 | 3 | 6 | 4 |
| **6** | 7 | 1 | 5 | 3 | 4 | 8 | **2** | 9 |
| **7** | 5 | 6 | **2** | 4 | 1 | 9 | 8 | **3** |
| 1 | **9** | 3 | 8 | 5 | 6 | 2 | **4** | 7 |
| 8 | **4** | **2** | 9 | 7 | **3** | **5** | **1** | 6 |

Figure 8: Clue matrix #2 and the solved puzzle

As mentioned earlier, puzzle #3 did not converge to integers given the initial constraints above, so a choice matrix was added to make the 50/50 choice between the two options and hopefully nudge the solution to converge on only integers. Below are the initial decimal output and the modified feedback puzzle with only integers. It is hard to tell, but if you look closely, there are cells in the decimal matrix which are 3.5, but change to 1 or 6 in the final integer solution. This confirms that the logic that the solver came down to two choices and couldn't make a choice is correct because the cell that makes 3.5 is $\{.5, 0, 0, 0, 0, .5, 0, 0, 0\}$ which have decimals in indices 1 and 6. The program ran twice until it found an integer solution. The first time it found the decimal matrix because there was no feedback information yet, then the second time it chose the first decimal square in location $[1, 2]$ to be a 1, which was correct and the problem converged on a solution.

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

Initial decimal solution

$\Downarrow$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **4** | 3.5 | 2 | 7 | **3** | 5 | **9** | 4.5 | **7** |
| 7 | 5 | 7 | 5 | **9** | **1** | 5.5 | **2** | 2.5 |
| 4.5 | **3** | 9 | 7 | 6 | 2 | 5 | **6** | 2.5 |
| 9 | **7** | **5** | 3 | 3.5 | **4** | 7 | 4.5 | 2 |
| 2 | **4** | 3.5 | **5** | 7 | **8** | 4.5 | **9** | 2 |
| 3 | 3.5 | 8 | **2** | 3.5 | 9 | **4** | **7** | 5 |
| 3.5 | **2** | 3.5 | 6 | 6 | 3 | 7 | **5** | 9 |
| 5 | **8** | 4 | **9** | **2** | 7 | 1 | 3 | 6 |
| **7** | 9 | **3** | 1 | **5** | 6 | 2 | 4 | **8** |

Forces a choice until it converges

$\Downarrow$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **4** | 1 | 2 | 6 | **3** | 5 | **9** | 8 | **7** |
| 6 | 5 | 7 | 8 | **9** | **1** | 3 | **2** | 4 |
| 8 | **3** | 9 | 7 | 4 | 2 | 5 | **6** | 1 |
| 9 | **7** | **5** | 3 | 6 | **4** | 8 | 1 | 2 |
| 2 | **4** | 1 | **5** | 7 | **8** | 6 | **9** | 3 |
| 3 | 6 | 8 | **2** | 1 | 9 | **4** | **7** | 5 |
| 1 | **2** | 6 | 4 | 8 | 3 | 7 | **5** | 9 |
| 5 | **8** | 4 | **9** | **2** | 7 | 1 | 3 | 6 |
| **7** | 9 | **3** | 1 | **5** | 6 | 2 | 4 | **8** |

Figure 9: Clue matrix #3, the decimal output, and the solved puzzle with feedback

The solution to clue matrix #4 was correct, so I will skip to clue matrix #5 which is similar to #3 in the way that it did not initially solve with an integer solution.

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

Initial decimal solution
$\Downarrow$

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| **8** | 3.5 | 5.5 | 7 | 3 | 2.5 | 4.5 | 4.5 | 6.5 |
| 6.5 | 3 | **3** | **6** | 3.5 | 8 | 1 | 7 | 7 |
| 3.5 | **7** | 5 | 5 | **9** | 2 | **2** | 8 | 4.5 |
| 5 | **5** | 3 | 6 | 4.5 | **7** | 8 | 3 | 3.5 |
| 3 | 4.5 | 6 | 5 | **4** | **5** | **7** | 9 | 1.5 |
| 7 | 5 | 6.5 | **1** | 7 | 5.5 | 5.5 | **3** | 4.5 |
| 3.5 | 4.5 | **1** | 5.5 | 7 | 6.5 | 4 | **6** | **8** |
| 3 | 4.5 | **8** | **5** | 2.5 | 5 | 9 | **1** | 7 |
| 5.5 | **9** | 7 | 5.5 | 4.5 | 3.5 | **4** | 3.5 | 2.5 |

Forces a choice until it converges
$\Downarrow$

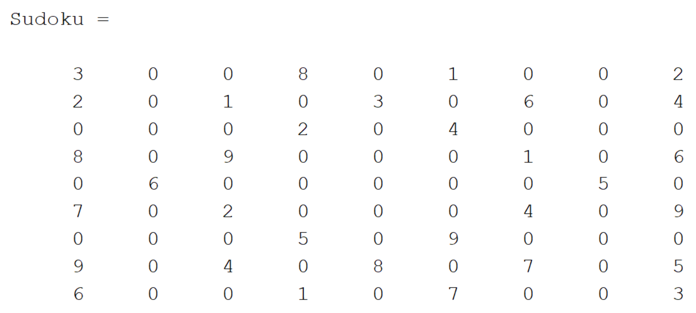
|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| **8** | 1 | 2 | 7 | 5 | 3 | 6 | 4 | 9 |
| 9 | 4 | **3** | **6** | 8 | 2 | 1 | 7 | 5 |
| 6 | **7** | 5 | 4 | **9** | 1 | **2** | 8 | 3 |
| 1 | **5** | 4 | 2 | 3 | **7** | 8 | 9 | 6 |
| 3 | 6 | 9 | 8 | **4** | **5** | **7** | 2 | 1 |
| 2 | 8 | 7 | **1** | 6 | 9 | 5 | **3** | 4 |
| 5 | 2 | **1** | 9 | 7 | 4 | 3 | **6** | **8** |
| 4 | 3 | **8** | **5** | 2 | 6 | 9 | **1** | 7 |
| 7 | **9** | 6 | 3 | 1 | 8 | **4** | 5 | 2 |

Figure 10: Clue matrix #3, the decimal output, and the solved puzzle with feedback

The method to solve clue matrix #5 is the same as the method used for clue matrix 3, where the squares with decimals represent squares that were in the fence between two integer values. Unlike clue matrix 3, it wasn't able to find a solution after only guessing the first square since clue matrix 5 is the hardest Sudoku in the world, it needs more assistance to converge on the single integer solution. The code was set up to solve the clue matrix by guessing 4 random squares that had decimal values in the initial output from *linprog*. The reason the squares are random is because simply going in the same order as the squares are numbered wouldn't be as effective at constraining the problem since squares 2 and 3 are in the same row and column. In other words, by randomly choosing squares to guess, the Sudoku is being constrained in more directions which should help with conversion. The Random squares with the two integer possibilities are then randomly chosen as being one of the two numbers, and the program runs again. This happens until the output from *linprog* contains only integers. Since this method is a brute force method with no way to record previous guesses, this isn't the more efficient method and does take quite a few iterations until it guesses each of the squares correctly resulting in an integer solution.

## 4 Bonus Playable Sudoku

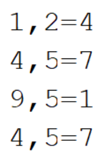
At the end of the program, once the matrices have been solved, it asks the user if they would like to play a game of Sudoku. The user selects "Yes" or "No" in a prompted dialogue box. If the player chooses not to, the program ends and nothing more happens. If the player chooses to play by hitting "Yes", they are prompted with a empty playing grid in the command window with a few starting squares filled in like a typical game of Sudoku. They aren't able to enter values to fill out the game, so the intention is they write it on paper and solve it there. The player is given the chance to try the game on their own, but if they'd like , a prompt pops up whether they'd like a hint or not. Again, the player can select "Yes" or "No". If they choose no, the prompt goes away and they have to solve the puzzle on their own. If they select yes, a hint pops up in the command window showing the location and value of one of the squares.



```
Sudoku =

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

Figure 11: Playing grid in the command window that the player can solve



```
1,2=4
4,5=7
9,5=1
4,5=7
```

Figure 12: Example of the hints in the command window. The form is matrix index on the left, and square value on the right