MCEN 5125: Optimal Design

Optimization Based Sudoku Solver

Jacob Haimes

1 Introduction

In 1783, the well known mathematician Leonhard Euler developed a game that, despite trends in some of his artifacts, was not named "Euler's Game" [1]. The puzzle was called Latin Squares, and it would be the predecessor to Number Place, a puzzle magazine game first published in 1979 by the New York based Dell Puzzle Magazine [2]. This brainteaser was modified once more when it travelled to Japan in 1984. Rebranded as Sudoku, the game gained quick popularity in Japan due to lack of puzzle competition (the written Japanese language does not translate well to a crossword format). Wayne Gould became aware of Sudoku while in vacation in Japan in 1997. He brought the idea of the Sudoku puzzle back to New York with him once his vacation was over. Over the next six years, Gould used much of his free time to write a computer program that would create a Sudoku puzzle [1, 2].

In this report, we will outline how to create a Sudoku puzzle solver that uses an optimization problem to find its solutions. The general form of a linear program will be reviewed in §2.1, followed by the rules of Sudoku, how to encode these rules in a linear program formulation for a 4×4 grid, and an extension to larger grids in §2.2, §2.3, and §2.4 respectively. The performance of our solver will be discussed in §3. Taking advantage of the advances in technology that have arisen since the turn of the century, we will attempt to create a program that generates a random, valid, and non-trivial Sudoku puzzle in §4. Provided our program does not take six years to write, our conclusions will follow in §5.

2 Formulation

Our goal in this paper is to create a program that will solve *most* Sudokus using a continuous domain linear programming optimization approach. Before we begin formulating our problem, however, it is important that we understand the general form of a linear program along with the rules that govern the puzzle we are trying to solve.

2.1 General Form for Linear Programming

Because we wish to solve our problem through linear programming, we first define the general form for a linear program:

$$\begin{aligned} \textbf{minimize} \quad & J^{\intercal} z && (1)\\ \textbf{subject to} \quad & Az \leq b && (2)\\ & A_{\text{eq}} z = b_{\text{eq}} && (3)\\ & \mathcal{L} \leq z \leq \mathcal{U} && (4) \end{aligned}$$

where $z \in \mathbb{R}^{\eta}$ is a column vector of the η variables, $J \in \mathbb{R}^{\eta}$ contains the coefficients of the cost function that we are trying to minimize, $A \in \mathbb{R}^{\mu \times \eta}$ is the μ constraint inequalities with $b \in \mathbb{R}^{\mu}$ representing the corresponding inequality constants, $A_{\text{eq}} \in \mathbb{R}^{\rho \times \eta}$ is the ρ constraint equations with $b_{\text{eq}} \in \mathbb{R}^{\rho}$ representing the corresponding inequality constants, and $\{\mathcal{L}, \mathcal{U}\} \in \mathbb{R}^{\eta}$ are the upper and lower bounds of the variables in z. Often, Eq. (1), Eq. (2), Eq. (3), and Eq. (4) are called the minimization function, inequality constraints, equality constraints, and bounds respectively. Once in this form, a linear program can be solved with a myriad of well documented solvers.

2.2 Rules of Sudoku

A Sudoku puzzle exists on an $N \times N$ grid of cells where $\{N \in \mathbb{N} \mid \texttt{ceil}(\sqrt{N}) = \texttt{floor}(\sqrt{N}) \wedge N > 1\}$, that is to say N must be a natural number greater than 1 whose square root is also a natural number. For cleanliness, we will denote

$$\mathcal{N} = N^2 \tag{5}$$

$$n = \sqrt{N}, \tag{6}$$

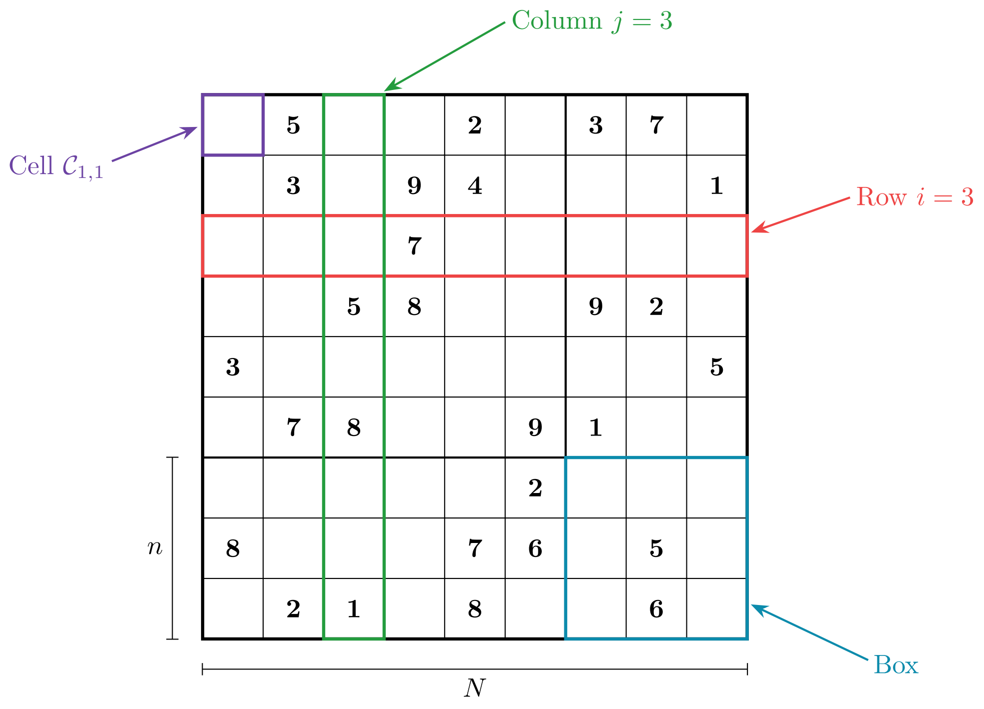


Figure 1: Annotated example of a 9×9 Sudoku grid. A significant number of cells are already filled in with digits, these are our *clue* cells. A Sudoku has been solved when each other cell is filled in with a digit from the set $\{1, 2, \ldots, 9\}$, and each row, column, and box has no duplicate digits.

as both n and $\mathcal{N}$ will turn out to be commonly used values throughout this process. Our grid is partitioned into N square boxes (each containing N cells). Our goal is to place a digit from the set $\{1, 2, \ldots, N\}$ in each cell such that all rows, columns, and boxes have exactly one of each possible digit. To specify our cells, we choose to use the same indexing scheme as is used in MATLAB for matrices; the cell in the top left of the grid is $\mathcal{C}^{\natural}_{1,1}$, with the the first index i increasing downwards, and the second index j increasing to the right. Although this representation is intuitive, we do need a one dimensional representation of these cells for our variable vector z. Using the index relations[1] between one dimensional and two dimensional array representations, we can write $\mathcal{C}^{\natural}_{N \times N}$ as an equivalent one dimensional vector $\mathcal{C}_{\mathcal{N} \times 1}$. Finally, some number of cells are already filled in with digits which cannot be changed from their current value. An annotated example Sudoku grid can be seen in Fig. (1).

2.3 The Optimization Problem

With the rules of our puzzle defined, we begin to contemplate how it can be rewritten as an optimization problem. To aid in this endeavor we will use a smaller-than-normal 4×4 Sudoku example, which can be seen in Fig. (2). Writing out the constraints and cost function for this more tractable problem will then allow us to easily see the patterns and assist us in our generalization of the formulation.

Obviously, the most pressing question that we must ask ourselves is how to mandate integers within each cell. We recall that the existence of an object can be considered a variable whose value is 0 if the object is not present, and positive otherwise. Potentially, we could give all cells $\mathcal{C}^{\natural}_{i,j} = C_k$ four different variables in our input vector z, one for each unique digit. A visualization of the z variables that

[1]An overview of the relations between one dimensional and two dimensional array representations can be seen in Appendix A.

		4	
1			
			3
	1		

Figure 2: Example 4×4 Sudoku that will be used to assist in our formulation of Sudoku solving as an optimization problem.

z_1	z_{17}	z_{33}	z_{49}
z_5	z_{21}	z_{37}	z_{53}
z_9	z_{25}	z_{41}	z_{57}
z_{13}	z_{29}	z_{45}	z_{61}

Figure 3: Our 4×4 example labeled with the z that governs digit 1 uniqueness per cell. This grid is used to assist in our constraints.

correspond to whether or not a cell contains the digit 1 can be seen in Fig. (3).

$$z = \begin{bmatrix} \mathcal{C}_{1\,(=1)} \\ \mathcal{C}_{1\,(=2)} \\ \mathcal{C}_{1\,(=3)} \\ \mathcal{C}_{1\,(=4)} \\ \mathcal{C}_{2\,(=1)} \\ \mathcal{C}_{2\,(=2)} \\ \vdots \\ \mathcal{C}_{N^2\,(=3)} \\ \mathcal{C}_{N^2\,(=4)} \end{bmatrix}, \tag{7}$$

which can also be expressed as the mapping

$$\mathcal{C}^{\natural}_{i,j\,(=\mathcal{D})} \to z_{\varkappa}, \qquad \text{where} \;\; \varkappa = (i + (j-1)N - 1)N + \mathcal{D}. \tag{8}$$

With this representation, we know that the true solution to a Sudoku will have the most zeros possible while still satisfying our row, column, and box constraints. In other words, we can find our solution by solving a maximization problem. This optimization problem is equivalent to minimizing the ℓ_0-norm, which unfortunately is a non-convex problem, meaning that we can't obtain a solution with continuous domain linear programming. Minimizing the ℓ_1-norm is rather similar to minimizing the ℓ_0-norm though, so potentially we can use this as a stand in for the ℓ_0-norm. Because our solver need not be infallible, we will press onward with this approach for now, knowing that if our solver behaves poorly, we will have to return to our problem formulation.

With our current definition of z, we examine the constraints imposed by the puzzle. Fundamentally, we want *uniqueness* of each digit in a specific set of cells. By design, our minimization function will attempt to maximize the number of zeros in z, allowing us to say that

$$\mathcal{C}^{\natural}_{1,1\,(=1)} + \mathcal{C}^{\natural}_{1,2\,(=1)} + \mathcal{C}^{\natural}_{1,3\,(=1)} + \mathcal{C}^{\natural}_{1,4\,(=1)} = 1 \tag{9}$$

is equivalent to mandating that exactly one of the cells in the first row contains the digit 1. We also pick up on the fact that the patterns for the uniqueness of 1 will be identical to the patterns of each other digit, but the values will be shifted upwards by 1, 2, or 3. We now write our constraints for digit 1 uniqueness using the linear indexing shown defined by Eq. (8) and shown in Fig. (3).

$$\begin{array}{rll}
\textbf{minimize} & \|z\|_1 & \\
\textbf{subject to} & z_1 + z_{17} + z_{33} + z_{49} = 1 & \text{Unique 1 - Row 1} \\
& z_5 + z_{21} + z_{37} + z_{53} = 1 & \text{Unique 1 - Row 2} \\
& z_9 + z_{25} + z_{41} + z_{57} = 1 & \text{Unique 1 - Row 3} \\
& z_{13} + z_{29} + z_{45} + z_{61} = 1 & \text{Unique 1 - Row 4} \\
& \left.\begin{array}{l} z_1 + z_5 + z_9 + z_{13} = 1 \\ z_{17} + z_{21} + z_{25} + z_{29} = 1 \\ z_{33} + z_{37} + z_{41} + z_{45} = 1 \\ z_{49} + z_{53} + z_{57} + z_{61} = 1 \end{array}\right\} & \text{Unique 1 - Columns} \\
& \left.\begin{array}{l} z_1 + z_5 + z_{17} + z_{21} = 1 \\ z_9 + z_{13} + z_{25} + z_{29} = 1 \\ z_{33} + z_{37} + z_{49} + z_{53} = 1 \\ z_{41} + z_{45} + z_{57} + z_{61} = 1 \end{array}\right\} & \text{Unique 1 - Boxes}
\end{array} \tag{10}$$

As mentioned earlier the constraints for each other digit, in this case 2, 3, and 4, have an identical pattern with each z index increased by 1, 2, or 3, respectively. We must also take into account that each cell $\mathcal{C}_k$ can only contain one digit. Eq. (11) shows the resulting set of constraints.

$$\begin{array}{rll}
\textbf{minimize} & \|z\|_1 & \\
\textbf{subject to} & \text{Unique 1 - Row 1} & \\
& \text{Unique 2 - Row 1} & \\
& \text{Unique 3 - Row 1} & \\
& \text{Unique 4 - Row 1} & \\
& \text{Unique 1 - Row 2} & \\
& \vdots & \\
& \text{Unique 4 - Row 4} & \\
& \text{Unique 1 - Column 1} & \\
& \vdots & \\
& \text{Unique 4 - Column 4} & \\
& \text{Unique 1 - Top Left Box} & \\
& \vdots & \\
& \text{Unique 4 - Bottom Right Box} & \\
& \left.\begin{array}{l} z_1 + z_2 + z_3 + z_4 = 1 \\ z_5 + z_6 + z_7 + z_8 = 1 \\ \vdots \\ z_{61} + z_{62} + z_{63} + z_{64} = 1 \end{array}\right\} & \text{1 Digit per Cell}
\end{array} \tag{11}$$

In our next iteration of our constraints, we will also account for the *clues* - the provided digits 1, 1, 4, and 3 in cells $\mathcal{C}^\natural_{2,1}$, $\mathcal{C}^\natural_{4,2}$, $\mathcal{C}^\natural_{1,3}$, and $\mathcal{C}^\natural_{3,4}$, respectively. We examine $\mathcal{C}^\natural_{2,1}$ first, recognizing that the relevant row, column, and box constraints are those pertaining to the clue digit: 1. We remove all other z variables from these equations as we know that other cells within the same row, column, or box as $\mathcal{C}^\natural_{2,1}$ cannot contain a 1.

$$\begin{array}{ll} z_5 + \cancel{z_{21}} + \cancel{z_{37}} + \cancel{z_{53}} = 1 & \text{Unique 1 - Row 2} \\ \cancel{z_1} + z_5 + \cancel{z_9} + \cancel{z_{13}} = 1 & \text{Unique 1 - Column 1} \\ \cancel{z_1} + z_5 + \cancel{z_{17}} + \cancel{z_{21}} = 1 & \text{Unique 1 - Top Right Box} \end{array} \tag{12}$$

In this manner we iteratively make modifications to our constraint equations for each *clue* cell. Once this is completed, we are almost ready to send our fully formulated linear program to a solver, but we have one more set of constraints to include: the upper and lower bounds on z. Until this point, we have assumed that the values within z are binary, but this is not the case. We will still attempt to get as close to this as possible though, with the constraints

$$0 \leq z \leq 1. \tag{13}$$

We have now reached a general form linear program, as outlined in §2.1. Note that although we do have both equality constraints and bounds, we do not have inequality constraints for this formulation.

2.4 Extension to Arbitrary Grid Size

The next step in our process is identifying the patterns in our 4×4 formulation such that we can extend it to any valid size Sudoku grid. We begin by investigating our constraint equations, some of which are written out in Eq. (10). Notice that each different type of constraint (row, column, or box) has a pattern that characterizes which z elements are in a constraint equation, which we can leverage to automate our constraint writing process. The ordering that we have chosen is useful here, as incrementing the index values of our patterns for digit 1 uniqueness result in the patterns for digit 2 uniqueness, and so on. This characteristic of the patterns is visualized with the row constraints in Eq. (14), while the patterns for column and box constraints can be viewed in Eq. (15) and Eq. (16), respectively.

$$\begin{array}{rl} & \quad\;\; \xrightarrow{16} \;\; \xrightarrow{16} \;\; \xrightarrow{16} \\ & z_1 + z_{17} + z_{33} + z_{49} = 1 \\ 4\downarrow & \\ & z_5 + z_{21} + z_{37} + z_{53} = 1 \\ 4\downarrow & \\ & z_9 + z_{25} + z_{41} + z_{57} = 1 \\ 4\downarrow & \\ & z_{13} + z_{29} + z_{45} + z_{61} = 1 \end{array} \quad \xrightarrow{i^{++}} \quad \begin{array}{rl} & \quad\;\; \xrightarrow{16} \;\; \xrightarrow{16} \;\; \xrightarrow{16} \\ & z_2 + z_{18} + z_{34} + z_{50} = 1 \\ 4\downarrow & \\ & z_6 + z_{22} + z_{38} + z_{54} = 1 \\ 4\downarrow & \\ & z_{10} + z_{26} + z_{42} + z_{58} = 1 \\ 4\downarrow & \\ & z_{14} + z_{30} + z_{46} + z_{62} = 1 \end{array} \tag{14}$$

$$\begin{array}{rl} & \quad\;\; \xrightarrow{4} \;\; \xrightarrow{4} \;\; \xrightarrow{4} \\ & z_1 + z_5 + z_9 + z_{13} = 1 \\ 16\downarrow & \\ & z_{17} + z_{21} + z_{25} + z_{29} = 1 \\ 16\downarrow & \\ & z_{33} + z_{37} + z_{41} + z_{45} = 1 \\ 16\downarrow & \\ & z_{49} + z_{53} + z_{57} + z_{61} = 1 \end{array} \tag{15}$$

$$\begin{array}{rl} & \quad\;\; \xrightarrow{4} \;\; \xrightarrow{16} \;\; \xrightarrow{4} \\ & z_1 + z_5 + z_{17} + z_{21} = 1 \\ 8\downarrow & \\ & z_9 + z_{13} + z_{25} + z_{29} = 1 \\ 24\downarrow & \\ & z_{33} + z_{37} + z_{49} + z_{53} = 1 \\ 8\downarrow & \\ & z_{41} + z_{45} + z_{57} + z_{61} = 1 \end{array} \tag{16}$$

For our row and column constraints, the patterns are rather simple, and have a clear relation to our grid size N. This means that for each digit $\mathcal{D} \in [1, \ldots, n]$ the row constraints are given by

$$\mathcal{D} + \begin{bmatrix} 0 & \mathcal{N} & 2\mathcal{N} & \cdots & (N-1)\mathcal{N} \\ N & N+\mathcal{N} & N+2\mathcal{N} & & \\ 2N & 2N+\mathcal{N} & 2N+2\mathcal{N} & & \\ \vdots & & & \ddots & \\ (N-1)N & & & & N^3-N \end{bmatrix} = \mathbb{1}_{(N\times 1)}, \tag{17}$$

while the column constraints are given by

$$\mathcal{D} + \begin{bmatrix} 0 & N & 2N & \cdots & (N-1)N \\ \mathcal{N} & \mathcal{N}+N & \mathcal{N}+2N & & \\ 2\mathcal{N} & 2\mathcal{N}+N & 2\mathcal{N}+2N & & \\ \vdots & & & \ddots & \\ (N-1)\mathcal{N} & & & & N^3-N \end{bmatrix} = \mathbb{1}_{(N\times 1)}. \tag{18}$$

In retrospect, we appreciate that the matrix in Eq. (17) is the transpose of the matrix in Eq. (18), just as rows are the transpose of columns. The box constraints do not appear to have a significant relation to our other patterns, but eventually we arrive at a method to create these constraints. First we define the column pattern in our box constraint matrix to be

$$\begin{aligned} \mathcal{P}_{\text{box,vert}} = & \begin{bmatrix} \overbrace{0 \;\; N \;\; \cdots \;\; (n-1)N}^{\text{set } 1} & \overbrace{0 \;\; N \;\; \cdots \;\; (n-1)N}^{\text{set } 2} & \cdots & \overbrace{0 \;\; N \;\; \cdots \;\; (n-1)N}^{\text{set } n} \end{bmatrix} \\ + & \begin{bmatrix} \overbrace{0 \;\; 0 \;\; \cdots \;\; 0}^{n \text{ times}} & \overbrace{\mathcal{N} \;\; \mathcal{N} \;\; \cdots \;\; \mathcal{N}}^{n \text{ times}} & \cdots & \overbrace{(n-1)\mathcal{N} \;\; \cdots \;\; (n-1)\mathcal{N}}^{n \text{ times}} \end{bmatrix}, \end{aligned} \tag{19}$$

and the row pattern to be

$$\begin{aligned} \mathcal{P}_{\text{box,horiz}} = & \begin{bmatrix} \overbrace{0 \;\; nN \;\; \cdots \;\; (n-1)nN}^{\text{set } 1} & \overbrace{0 \;\; nN \;\; \cdots \;\; (n-1)nN}^{\text{set } 2} & \cdots & \overbrace{0 \;\; nN \;\; \cdots \;\; (n-1)nN}^{\text{set } n} \end{bmatrix}^{\intercal} \\ + & \begin{bmatrix} \overbrace{0 \;\; 0 \;\; \cdots \;\; 0}^{n \text{ times}} & \overbrace{n\mathcal{N} \;\; n\mathcal{N} \;\; \cdots \;\; n\mathcal{N}}^{n \text{ times}} & \cdots & \overbrace{(n-1)n\mathcal{N} \;\; \cdots \;\; (n-1)n\mathcal{N}}^{n \text{ times}} \end{bmatrix}^{\intercal}. \end{aligned} \tag{20}$$

With Eq. (19) and Eq. (20) defined, we can write an equation similar to Eq. (17) and Eq. (18) for our box constraints:

$$\mathcal{D} + \texttt{repmat(}\mathcal{P}_{\text{box,vert}}\texttt{,}N\texttt{,1)} + \texttt{repmat(}\mathcal{P}_{\text{box,horiz}}\texttt{,1,}N\texttt{)} = \mathbb{1}_{(N\times 1)}, \tag{21}$$

where the function `repmat(mat,d1,d2)` repeats the input matrix `mat` along the vertical dimension `d1` times, and the horizontal dimension `d2` times[2].

3 Results

With our formulation finished, we must attempt to verify it. MATLAB code that writes our constraint equations based on an input clue grid of any valid Sudoku size can be viewed in Appendix B. CVX, a

[2]The `repmat` function is a common function used in MATLAB. Mathworks documentation for the `repmat` function can be found here.

1	**5**	9	6	**2**	8	**3**	**7**	4
7	**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 4: The solution generated by our solver for the example Sudoku grid seen in Fig. (1).

MATLAB-based convex modeling framework was then implemented to solve our linear program using the open source SeDuMi solver [3].

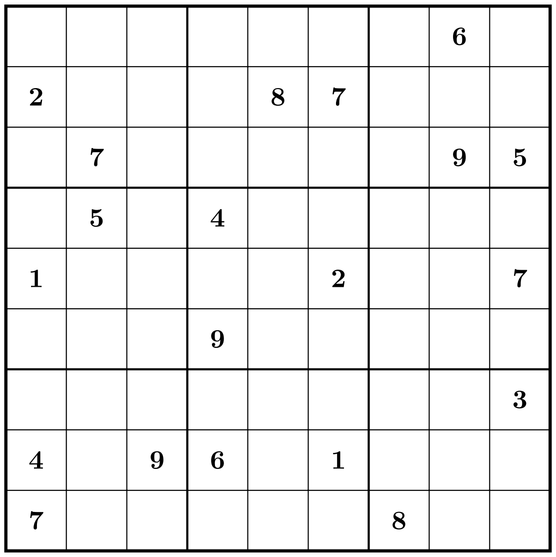
To assess the solver, we use it on four different Sudoku puzzles of increasing difficulty. The precise inputs for our framework can be found in Appendix B. For the first three puzzles, our solver works very well, generating the solution and printing it in just over half a second. For the fourth Sudoku our solver still returns an answer quickly, but it is not one that can be interpreted as a Sudoku grid, as the z vector contains many entries in between 1 and 0. This, however, is somewhat expected. Recall that we are *not* using the ℓ_0 norm, but we are instead using the ℓ_1 norm, as calculating the ℓ_0 norm is not a problem that can be solved through traditional linear programming techniques. Here we have made the compromise of solver completeness to simplify our problem.

4 Generating a Sudoku

To expand on this problem we will attempt to generate a valid Sudoku puzzle by utilizing the solver that we have made. To do this, we first choose a random index within our z vector, which can be accomplished by generating a random integer $d \in [1, N^3]$. Note that d provides both the location of the relevant cell $\mathcal{C}_{i,j}$ as well as the digit within that cell $\mathcal{D}_1$. We then initialize $\mathcal{C}$ as an $N \times N$ matrix of zeros, and set $\mathcal{C}_{i,j} = \mathcal{D}_1$.

So far, we have effectively chosen a random cell on an empty $N \times N$ grid, and filled it with digit $\mathcal{D}_1$. Although we know that there are many possible solutions to this particular puzzle, we can still create a linear program following the process described in §2.3 and §2.4, and solve it. We know that exactly one index in z will contain a one, as our constraints mandated that $z_d = 1$. We can also obtain valuable information from the zero elements within z, as these can be interpreted as *impossible* configurations; if the the value in these is anything but zero, our puzzle is breaking the rules of Sudoku.

Using this insight, we can randomly choose the next component of our clue from the pool of indices that do not contain a zero or a one, ensuring that the resulting clue can still be solved in a manner consistent with the Sudoku rule-set. Constraints for a new LP are written, and the LP is solved in the same manner as before. After repeating this process multiple times, we may eventually arrive at a scenario in which a value in z is close to, but not exactly, zero or one. This can be interpreted as constraints implied by our clue. It is important that we also remove these from our pool of indices as well.



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

Figure 5: Example of a valid Sudoku that was created using our optimization based generation method. Specifically, the 'near min' variant was used for this puzzle.

Once all indices are either very close to zero or very close to one, the program terminates, outputting our clue matrix. At this point we must test the output clue matrix, but it would probably be a good idea to use an unbiased third party (i.e. we shouldn't validate the clue matrix made by our solver with that very solver). For this reason, we utilize the non-optimization based Sudoku solver by dCode for validation [5].

There are three distinct outcomes for our generated puzzle: a valid Sudoku, an impossible Sudoku, or an invalid Sudoku. The `sudoku_gen` function will recognize when it has created an impossible Sudoku, and will report the clue matrix prior to the inclusion of the constraint that made the Sudoku impossible. In these cases, the puzzles returned are not unique, but a few more correctly placed constraints should result in a valid clue. An invalid Sudoku, on the other hand, is a little bit more difficult to identify. To characterize the invalid clues that can be created with our method we will use another 4×4 example, seen in Fig. (6). In this example, we have fully defined two of the four digits, but the remaining two digits are minimally constrained (within the remaining cells). This results in a puzzle with many solutions, meaning that it cannot be a valid Sudoku. Luckily, the scenario is very unlikely to occur, meaning that practically every puzzle that we generate that is not impossible will be valid.

In addition to purely random generation of our clues, we may explore different methods with which to define which indices in z we want to fix. Interestingly, choosing d such that

$$\mathbf{min}\Big(\big\{z \mid z_k \neq 0\big\}\Big) \approx z_d \tag{22}$$

turns out to be an effective way of reducing the number of defined cells in the clue grid.

5 Conclusion

We recall that the purpose of this report was to design a Sudoku solver that utilizes linear programming to find the solution to *most* Sudoku puzzles. To formulate our problem, we first solved a smaller problem in order to identify patterns within our solution. With these patterns identified, they were generalized to work for any $N \times N$ grid, so long as $n = \sqrt{N} \in \mathbb{Z}$. Following our generalization, the entirety of our process was implemented in MATLAB and tested against multiple Sudoku puzzles. Although not able to solve the hardest of these example puzzles, all others were solved efficiently.

		2	
	1		
		1	
	2		

(a) Clue

3 or 4	3 or 4	**2**	1
2	**1**	3 or 4	3 or 4
3 or 4	3 or 4	**1**	2
1	**2**	3 or 4	3 or 4

(b) Partial solution

Figure 6: Example 4×4 clue that is *not* a valid Sudoku puzzle (a) and it's partial solution (b). Due to the unique geometry of this clue, two of our digits are fully defined, while the remaining two are is ill defined as possible.

Finally, we used our Sudoku solver to build a Sudoku clue from a blank grid. Our method of Sudoku puzzle generation is not flawless, and it cannot create puzzles that our optimization method cannot solve, but it can create interesting Sudoku puzzles in a novel way with a decent success rate. One particularly interesting discovery was that we could create more sparse Sudoku puzzles by selecting which z index d would be fixed next from the smallest non-zero values within the previous z vector.

It would be interesting to explore the clue-generator aspect of this project more in depth. I think that it would be attainable to create a generator that had no chance of returning an invalid or impossible solution (provided that the puzzle could be solved with our methods). Another interesting direction to take this project would be to determine what modifications to our process would be necessary to make our solver work for any Sudoku.

References

[1] D. Smith, Ed., "So you thought Sudoku came from the Land of the Rising Sun," The Guardian, 15-May-2005. [Online]. Available: https://www.theguardian.com/media/2005/may/15/pressandpublishing.usnews.

[2] "The History of Sudoku," *sudoku.com*. [Online]. Available: https://sudoku.com/how-to-play/the-history-of-sudoku/.

[3] Michael Grant and Stephen Boyd. *CVX: Matlab software for disciplined convex programming*, version 3.0 beta. http://cvxr.com/cvx, December 2017.

[4] Emerging Technology from the arXiv. "Mathematicians solve minimum Sudoku problem," *MIT Technology Review*, 02-Apr-2020. [Online]. Available: https://www.technologyreview.com/2012/01/06/188520/mathematicians-solve-minimum-sudoku-problem/.

[5] dCode, *dCode*, 2022. [Online]. Available: https://www.dcode.fr/.

[6] R. Bonvallet, "Example: Sudoku," *TeXample.net*, 01-Feb-2012. [Online]. Available: https://texample.net/tikz/examples/sudoku/.

A 1D - 2D Index Relations

When converting from a one dimensional array, P_{m^2}, to a two dimensional array, $Q_{m\times m}$, Eq. (23) can be used to determine corresponding indices[3]. Similarly, Eq. (24) can be used to convert from two dimensional indices to one dimensional indices. A visual representation of the pixel locations (v and w in an $m\times m$ matrix) their corresponding index in the provided form (u in a $1\times n^2$ array) is given in Eq. (25).

$$\begin{aligned} &g : P_u \to Q_{v,w} \\ &\text{where } \; g(u) = (v,w) = \left((u-1 \pmod m) + 1, \quad \left\lfloor \frac{u-1}{m} \right\rfloor + 1 \right) \end{aligned} \tag{23}$$

$$\begin{aligned} &h : Q_{v,w} \to P_u \\ &\text{where } \; h(v,w) = u = m(w-1) + v \end{aligned} \tag{24}$$

$$\begin{bmatrix} 1 & m+1 & 2m+1 & \cdots & m(m-1)+1 \\ 2 & m+2 & 2m+2 & \cdots & m(m-1)+2 \\ 3 & m+3 & 2m+3 & \cdots & m(m-1)+3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m & 2m & 3m & \cdots & m^2 \end{bmatrix} \tag{25}$$

[3]Eq. (23) and Eq. (24) are only true when converting between arrays and matrices whose indexing begins at 1.

B Matlab Code

B.1 Provided Sudoku Matrices

```
% MEDIUM LEVEL
sudoku_mat1 = [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
sudoku_mat2 = [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
sudoku_mat3 = [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
sudoku_mat4 = [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];
```

B.2 Main Runner

```
% Load in sudokus
sudokus

% Attempt to solve all provided puzzles, print them if possible
sudoku_sol1 = sudoku_solve(sudoku_mat1);
print_sudoku(sudoku_sol1,9)

sudoku_sol2 = sudoku_solve(sudoku_mat1);
print_sudoku(sudoku_sol2,9)

sudoku_sol3 = sudoku_solve(sudoku_mat1);
print_sudoku(sudoku_sol3,9)

sudoku_sol4 = sudoku_solve(sudoku_mat1);
print_sudoku(sudoku_sol4,9)

% The keyboard function is used multiple times in the remainder of this
% file to pause execution until progressed by the user. This is because the
% MATLAB command window will be cleared during the following function call.
keyboard

rand_gen_sudoku = sudoku_gen(9);
print_sudoku(rand_gen_sudoku,9)

keyboard
nmin_gen_sudoku = sudoku_gen(9,'near min');
print_sudoku(nmin_gen_sudoku,9)

keyboard
tmin_gen_sudoku = sudoku_gen(9,'true min');
print_sudoku(tmin_gen_sudoku,9)

keyboard
nmax_gen_sudoku = sudoku_gen(9,'near max');
print_sudoku(nmax_gen_sudoku,9)
```

B.3 Solve Sudoku

```
function sol = sudoku_solve(clue)
%SUDOKU_SOLVE

% Set up A and b for constraint equations
[A,N] = constrain(clue);
b = ones(4*N^2,1);

% Solve the linear program
cvx_begin quiet
    variable z(N^3) nonnegative
    minimize ( norm(z,1) )
    subject to
        A*z == b
        z <= 1
cvx_end

% Filter out really small values
z(z<1e-3 & z>-1e-3) = 0;

% Return NaN if our solver did not find a viable Sudoku solution
if sum(z > 0) ~= N^2
    warning("Minimizing the one norm did not result in a solution to the Sudoku
         puzzle.")
    sol = NaN;
    return
end

% Determine digit for each cell
C = zeros(N^2,1);
for i = 1:N^2
    C(i) = find(z((N*(i-1)+1):(N*(i-1)+N)));
end

% Reshape into the appropriate dimension
sol = reshape(C,N,[]);

end
```

B.4 Define LP Constraints

```
function [A,N] = constrain(clue)
%CONSTRAIN

% Verify that input is a square matrix of size N, where N is a perfect
% square greater than 1.
[N,N2] = size(clue);
n = sqrt(N);
if N ~= N2 || ( ceil(n) ~= floor(n) ) || N < 4
    error('Please input a square matrix whose number of row is a perfect square
        .')
end
clear('N2');

% Create general row constraints
%   -> There can only be one of each digit in each row
row_base = 1:N^2:((N-1)*N^2+1);          % A matrix horizontal pattern
row_skip = (0:N:N^2-1)';                  % A matrix vertical pattern

row_cons = arrayfun(@(a) ...              % Combine + replicate patterns per
    digit
    repelem(row_base,N,1) + repelem(row_skip,1,N) + a, ...
    0:(N-1),'UniformOutput',false);

% Create general column constraints
%   -> There can only be one of each digit in each column
col_base = 1:N:((N-1)*N+1);              % A matrix horizontal pattern
col_skip = (0:N^2:N^3-1)';               % A matrix vertical pattern

col_cons = arrayfun(@(a) ...              % Combine + replicate patterns per
    digit
    repelem(col_base,N,1) + repelem(col_skip,1,N) + a, ...
    0:(N-1),'UniformOutput',false);

% Create general box constraints
%   -> There can only be one of each digit in each box
box_1dim = linspace(1,1+(n-1)*N,n);
box_1dim = repmat(box_1dim,1,n);
box_2dim = linspace(1,1+(n-1)*N^2,n)-1;
box_2dim = repelem(box_2dim,1,n);
box_base = box_1dim + box_2dim;          % A matrix horizontal pattern

box_skp1 = linspace(0,n*(n-1)*N,n)';
box_skp1 = repmat(box_skp1,n,1);
box_skp2 = linspace(0,n*(n-1)*N^2,n)';
box_skp2 = repelem(box_skp2,n,1);
box_skip = box_skp1 + box_skp2;          % A matrix vertical pattern

box_cons = arrayfun(@(a) ...              % Combine + replicate patterns per
    digit
    repelem(box_base,N,1) + repelem(box_skip,1,N) + a, ...
    0:(N-1),'UniformOutput',false);
```

⋮

⋮

```
% Create general A matrix
A = zeros(3*N^2,N^3);
NN = N^2;
for i = 1:NN
    A(       i, row_cons{ ceil(i/N) }( mod(i-1,N)+1,: ) ) = 1;
    A(   N^2+i, col_cons{ ceil(i/N) }( mod(i-1,N)+1,: ) ) = 1;
    A( 2*N^2+i, box_cons{ ceil(i/N) }( mod(i-1,N)+1,: ) ) = 1;
end

clue_bool = clue ~= 0;
clue_num = sum(clue_bool,'all');
clue_locs = find(clue_bool);
A_temp = A;
for i = 1:clue_num
    clue_ind = clue_locs(i);
    clue_val = clue(clue_ind);
    relevant_eqs = repelem(find(A_temp(:,1+(clue_ind-1)*N)),N,1) + repmat((0:N
        :(N^2-1))',3,1);
    clue_eqs = A(relevant_eqs(clue_val + [0 N 2*N]),:);
    mask = clue_eqs(1,:) & clue_eqs(2,:) & clue_eqs(3,:);
    mask = repmat(mask,3,1) & A(relevant_eqs(clue_val + [0 N 2*N]),:);
    A(relevant_eqs(clue_val + [0 N 2*N]),:) = mask;
end

% Add constraint that each cell can only have one digit present
A = [ A; repelem(diag(ones(N^2,1)),1,N) ];

end
```

B.5 Generate New Sudoku

```
function sudoku = sudoku_gen(N,gen_type)
%SUDOKU_GEN

dig = randi([1,N^3]);
cell = ceil(dig/N);
val = mod(dig-1,N)+1;

sudoku = zeros(N);
sudoku(cell) = val;

it = 0;
[change,z,time] = sudoku_try(sudoku);

it = it + 1;
cum_time = time;
print_update(it,time,cum_time)

if ~exist('gen_type','var') || isempty(gen_type) ...
        || any(strcmp(gen_type,{'random','r','rand','default'}))
    % Random
    while any(change)
        choice = change .* (1:N^3)';
        choice(abs(choice) <= .001) = [];
        ind = randi([1,sum(change)]);
        dig = choice(ind);
        cell = ceil(dig/N);
        val = mod(dig-1,N)+1;
        sudoku(cell) = val;
        z_temp = z;
        [change,z,time] = sudoku_try(sudoku);
        it = it + 1;
        cum_time = time + cum_time;
        print_update(it,time,cum_time)
    end
elseif any(strcmp(gen_type,{'near min'}))
    % Choose from current min z values
    while any(change)
        z_temp = z;
        z_temp(z==0) = inf;
        choice = find(abs(z_temp-min(z_temp))<.01);
        ind = randi([1,numel(choice)]);
        dig = choice(ind);
        cell = ceil(dig/N);
        val = mod(dig-1,N)+1;
        sudoku(cell) = val;
        [change,z,time] = sudoku_try(sudoku);
        it = it + 1;
        cum_time = time + cum_time;
        print_update(it,time,cum_time)
    end

```

⋮

⋮

```
elseif any(strcmp(gen_type,{'true min'}))
    % Choose from current min z values
    while any(change)
        z_temp = z;
        z_temp(z==0) = inf;
        choice = find(abs(z_temp-min(z_temp))<.0001);
        ind = randi([1,numel(choice)]);
        dig = choice(ind);
        cell = ceil(dig/N);
        val = mod(dig-1,N)+1;
        sudoku(cell) = val;
        [change,z,time] = sudoku_try(sudoku);
        it = it + 1;
        cum_time = time + cum_time;
        print_update(it,time,cum_time)
    end
elseif any(strcmp(gen_type,{'near max'}))
    % Choose from current max z values
    while any(change)
        z_temp = z;
        z_temp(z==0) = -inf;
        z_temp(z==1) = -inf;
        choice = find(abs(z_temp-max(z_temp))<.01);
        ind = randi([1,numel(choice)]);
        dig = choice(ind);
        cell = ceil(dig/N);
        val = mod(dig-1,N)+1;
        sudoku(cell) = val;
        [change,z,time] = sudoku_try(sudoku);
        it = it + 1;
        cum_time = time + cum_time;
        print_update(it,time,cum_time)
    end
else
    error('The input variable gen_type was specified incorrectly.')
end

if any(isnan(z))
    [row,col] = ind2sub([N,N],cell);
    fprintf('\nCulprit\n\tDigit: (%i)\n\tCell: (%i,%i)\n\tz Value: %g\n\n',...
        val,row,col,z_temp(dig))
    warning(['Final clue definition resulted in impossible ' ...
        '(or highly difficult) Sudoku. Returning penultimate clue matrix.'])
    sudoku(cell) = 0;
    fprintf(['NOTE: THE BELOW IS NOT A VALID SUDOKU ' ...
             'AND MOST LIKELY HAS MANY POSSIBLE SOLUTIONS\n\n'])
end

if N == 9
    condense(sudoku);
elseif N == 16
    hexify(sudoku);
end

end
```

B.6 Modified Solution Function for Sudoku Generation

```
function [change,z,cvx_cputime] = sudoku_try(clue)
%SUDOKU_TRY

% Set up A and b for constraint equations
[A,N] = constrain(clue);
b = ones(4*N^2,1);

% Solve the linear program
cvx_begin quiet
    variable z(N^3) nonnegative
    minimize ( norm(z,1) )
    subject to
        A*z == b
        z <= 1
cvx_end

% Filter out really small values
z(z<1e-3 & z>-1e-3) = 0;

% Return NaN if our solver did not find a viable Sudoku solution
if sum(z > 0) ~= N^2
    change = (z>0 & z<1);
else
    change = false;
end

end
```

B.7 Print Update for Sudoku Generation

```
function print_update(it,prev,tot)
%PRINT_UPDATE
clc;
fprintf(['\n\t\t\t************************************************\n' ...
         '\t\t\t    Iteration %i Completed \n\n' ...
         '\t\t\t    Previous solution time: %.2f seconds\n' ...
         '\t\t\t    Total CPU time: %.2f seconds\n\n' ...
         '\t\t\t************************************************\n\n'], ...
         it,prev,tot);
end
```

B.8 Prep 9x9 Sudoku for dCode Solver

```
function oneline = condense(in_sudoku)
%CONDENSE

    oneline = string(in_sudoku);
    oneline(oneline == "0") = " ";
    oneline = oneline';
    oneline = reshape(char(oneline),1,[]);
    fprintf(['**********\nInput solution line into solver at ' ...
             'https://www.dcode.fr/sudoku-solver for typical solver ' ...
             'verification.\n\tDo not use the "FILL" feature (it removes ' ...
             'leading spaces)\n\tInstead, select the top left cell and ' ...
             'press Ctrl+V\n\nSolution line:\n'])
    fprintf([oneline '\n**********\n\n']);
end
```

B.9 Prep 16x16 Sudoku for dCode Solver

```
function hexed = hexify(in_sudoku)
%HEXED

    hexed = string(in_sudoku);
    hexed(hexed == "10") = "A";
    hexed(hexed == "11") = "B";
    hexed(hexed == "12") = "C";
    hexed(hexed == "13") = "D";
    hexed(hexed == "14") = "E";
    hexed(hexed == "15") = "F";
    hexed(hexed ==  "0") = " ";
    hexed(hexed == "16") = "0";
    hexed = hexed';
    hexed = reshape(char(hexed),1,[]);
    fprintf(['**********\nInput solution line into solver at ' ...
             'https://www.dcode.fr/hexadoku-sudoku-16-solver for typical ' ...
             'solver verification.\n\tDo not use the "FILL" feature (it ' ...
             'removes leading spaces)\n\tInstead, select the top left ' ...
             'cell and press Ctrl+V\n\nSolution line:\n'])
    fprintf([hexed '\n**********\n\n']);
end
```