Algorithmic Solving of Sudoku Puzzles

Mr. Lukas Gribbin
Project Inspired by Dr. James Bailey

Version 1.1

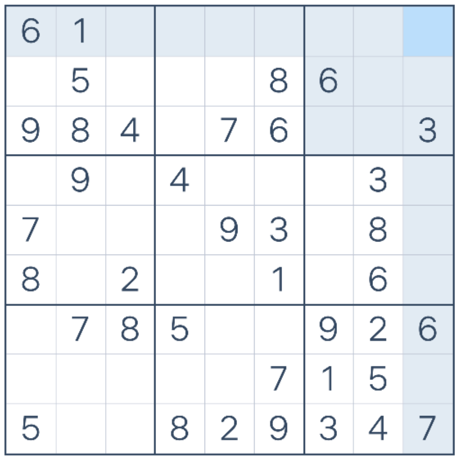
Contents

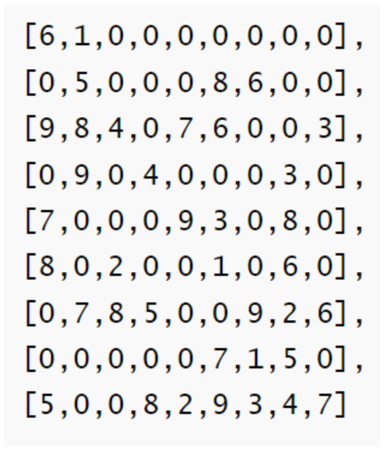
1 Introduction – Describing the Problem

A Sudoku puzzle is represented as a 9×9 grid. Empty cells are denoted by a 0, while pre-filled cells contain their respective numbers. The solver's task is to fill the empty cells with digits from 1 to 9, adhering to the following rules:

1. Each row must contain every digit from 1 to 9 exactly once.
2. Each column must contain every digit from 1 to 9 exactly once.
3. Each of the nine 3×3 subgrids must contain every digit from 1 to 9 exactly once.

The initial configuration of numbers determines the puzzle's difficulty. Figure 1 illustrates a sample puzzle and its corresponding array representation.





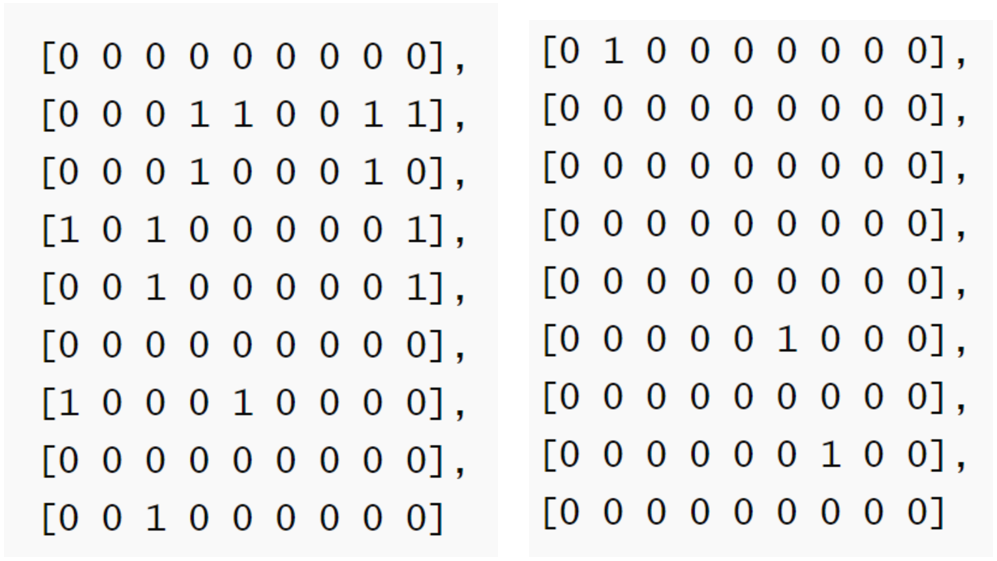
```
[6,1,0,0,0,0,0,0,0],
[0,5,0,0,0,8,6,0,0],
[9,8,4,0,7,6,0,0,3],
[0,9,0,4,0,0,0,3,0],
[7,0,0,0,9,3,0,8,0],
[8,0,2,0,0,1,0,6,0],
[0,7,8,5,0,0,9,2,6],
[0,0,0,0,0,7,1,5,0],
[5,0,0,8,2,9,3,4,7]
```

Figure 1: A Sudoku puzzle and its array representation.

2 Solving Process

2.1 CreateMask Function

The core of our solving method is a function that generates a "mask" for each digit from 1 to 9. A mask for a digit x is a binary 9×9 array where a 1 indicates a possible location for x and a 0 indicates an invalid location. Figure 2 shows an example of a mask for the digit 1.



```
[0 0 0 0 0 0 0 0 0],
[0 0 0 1 1 0 0 1 1],
[0 0 0 1 0 0 0 1 0],
[1 0 1 0 0 0 0 0 1],
[0 0 1 0 0 0 0 0 1],
[0 0 0 0 0 0 0 0 0],
[1 0 0 0 1 0 0 0 0],
[0 0 0 0 0 0 0 0 0],
[0 0 1 0 0 0 0 0 0]
```

```
[0 1 0 0 0 0 0 0 0],
[0 0 0 0 0 0 0 0 0],
[0 0 0 0 0 0 0 0 0],
[0 0 0 0 0 0 0 0 0],
[0 0 0 0 0 0 0 0 0],
[0 0 0 0 0 1 0 0 0],
[0 0 0 0 0 0 0 0 0],
[0 0 0 0 0 0 1 0 0],
[0 0 0 0 0 0 0 0 0]
```

Figure 2: The mask for the digit 1 (left) and the locations of 1s in the puzzle (right).

The process for creating a mask for a digit x is as follows:

1. Initialize an empty mask array.
2. For each instance of the digit x already in the puzzle:
   (a) Mark its entire row as invalid (0) in the mask.
   (b) Mark its entire column as invalid (0) in the mask.
   (c) Mark its entire 3×3 subgrid as invalid (0) in the mask.
3. Mark all cells that are already occupied by any number as invalid (0).

The Python function, `CreateMasks`, which implements this logic, is shown in Listing 1.

```python
# Creates 9 masks for possible positions of digits 1-9
def createMasks(puzzle):
    masks = []
    a = np.ones([3, 3])
    for i in range(1,10):
        mask = ((np.ones((9,9)) * i) == puzzle) * 1
        for index in np.argwhere(mask):
            # SET ROW
            mask[index[0],:] = 1
            # SET COLUMN
            mask[:,index[1]] = 1
            # SET 3X3 SQUARES
            a0, a1 = (index[0]//3)*3, (index[1]//3)*3
            mask[a0:a0+a.shape[0],a1:a1+a.shape[1]] = a
        # SET EXISTING NUMBERS IN PUZZLE
        mask[puzzle != 0] = 1
        masks.append(np.absolute(mask-1))
    return np.array(masks)
```

Listing 1: CreateMask Function Python Code

2.2 SolveKnown Function

With the masks generated for each digit, we can identify cells that have only one possible value. We achieve this by summing all nine masks together into an ”overlapped array.” Each element in this array represents the total number of possible digits for that cell. If a cell has a value of 1, it means it is immediately solvable.

```
Overlapped Array:
[0 0 2 3 3 3 5 2 5]
[2 0 2 4 3 0 0 3 4]
[0 0 0 2 0 0 2 1 0]
[1 0 3 0 3 2 3 0 3]
[0 2 3 2 0 0 3 0 4]
[0 2 0 1 1 0 3 0 3]
[3 0 0 0 3 1 0 0 0]
[3 4 3 2 3 0 0 0 1]
[0 1 2 0 0 0 0 0 0]
```

The `SolveKnown` function finds the positions of all 1s in the overlapped array. For each such position, it checks the nine masks to determine which digit corresponds to that single possibility and updates the puzzle accordingly. The code is shown in Listing 3.

```python
# Solves known empty places using the masks
def solveKnown(masks):
    arraySum = np.sum(masks,0)
    solveIndex = np.where(arraySum == 1)
    puzzle[solveIndex[0],solveIndex[1]] = np.where(masks[:,solveIndex[0],solveIndex[1]].T)[1]+1
    return puzzle
```

Listing 2: SolveKnown Function Python Code

2.3 Putting it All Together

The complete solving process involves alternating between creating masks and solving for known values until the puzzle is complete.

1. Generate the masks for the initial puzzle state.
2. Use the masks to solve for all uniquely determined cells.
3. Repeat steps 1 and 2 with the updated puzzle.
4. Continue this iterative process until no more zeros remain in the puzzle.

Figure 3 shows the puzzle's state through several iterations of the solving algorithm.

And the following code shows how these steps are implemented:

```python
# Runs the functions until the puzzle is complete
i = 0
while True:
    puzzle = solveKnown(createMasks(puzzle))
    i = i + 1
    if np.sum(puzzle==0) == 0 or i > 80:
        print(f"Puzzle solved in {i} steps:")
        print(puzzle)
        break
```

Listing 3: SolveKnown Function Python Code

```
Original                  Step 1                  Step 2
[6 1 0 0 0 0 0 0 0]   [6 1 0 0 0 0 0 0 0]   [6 1 0 0 0 0 0 0 0]
[0 5 0 0 0 8 6 0 0]   [0 5 0 0 0 8 6 0 0]   [0 5 0 0 0 8 6 0 0]
[9 8 4 0 7 6 0 0 3]   [9 8 4 0 7 6 0 1 3]   [9 8 4 2 7 6 0 1 3]
[0 9 0 4 0 0 0 3 0]   [1 9 0 4 0 0 0 3 0]   [1 9 0 4 0 2 0 3 0]
[7 0 0 0 9 3 0 8 0]   [7 0 0 0 9 3 0 8 0]   [7 4 0 0 9 3 0 8 0]
[8 0 2 0 0 1 0 6 0]   [8 0 2 7 5 1 0 6 0]   [8 0 2 7 5 1 4 6 0]
[0 7 8 5 0 0 9 2 6]   [0 7 8 5 0 4 9 2 6]   [3 7 8 5 0 4 9 2 6]
[0 0 0 0 0 7 1 5 0]   [0 0 0 0 0 7 1 5 8]   [0 0 0 0 0 7 1 5 8]
[5 0 0 8 2 9 3 4 7]   [5 6 0 8 2 9 3 4 7]   [5 6 1 8 2 9 3 4 7]

Step 3                    Step 4                  Step 5
[6 1 0 0 0 5 0 0 0]   [6 1 0 0 0 5 0 0 0]   [6 1 0 9 0 5 8 0 2]
[2 5 0 0 0 8 6 0 0]   [2 5 0 0 0 8 6 0 4]   [2 5 0 0 3 8 6 0 4]
[9 8 4 2 7 6 5 1 3]   [9 8 4 2 7 6 5 1 3]   [9 8 4 2 7 6 5 1 3]
[1 9 0 4 0 2 0 3 5]   [1 9 6 4 8 2 7 3 5]   [1 9 6 4 8 2 7 3 5]
[7 4 0 6 9 3 0 8 0]   [7 4 5 6 9 3 2 8 0]   [7 4 5 6 9 3 2 8 1]
[8 3 2 7 5 1 4 6 9]   [8 3 2 7 5 1 4 6 9]   [8 3 2 7 5 1 4 6 9]
[3 7 8 5 1 4 9 2 6]   [3 7 8 5 1 4 9 2 6]   [3 7 8 5 1 4 9 2 6]
[0 2 9 0 0 7 1 5 8]   [4 2 9 3 0 7 1 5 8]   [4 2 9 3 6 7 1 5 8]
[5 6 1 8 2 9 3 4 7]   [5 6 1 8 2 9 3 4 7]   [5 6 1 8 2 9 3 4 7]

Step 6                    Step 7 (Solved!)
[6 1 0 9 4 5 8 7 2]   [6 1 3 9 4 5 8 7 2]
[2 5 7 1 3 8 6 0 4]   [2 5 7 1 3 8 6 9 4]
[9 8 4 2 7 6 5 1 3]   [9 8 4 2 7 6 5 1 3]
[1 9 6 4 8 2 7 3 5]   [1 9 6 4 8 2 7 3 5]
[7 4 5 6 9 3 2 8 1]   [7 4 5 6 9 3 2 8 1]
[8 3 2 7 5 1 4 6 9]   [8 3 2 7 5 1 4 6 9]
[3 7 8 5 1 4 9 2 6]   [3 7 8 5 1 4 9 2 6]
[4 2 9 3 6 7 1 5 8]   [4 2 9 3 6 7 1 5 8]
[5 6 1 8 2 9 3 4 7]   [5 6 1 8 2 9 3 4 7]
```

Figure 3: Progression of the puzzle solution over 7 steps.

3 Conclusions

In this project, we successfully implemented a vectorized approach to solving Sudoku puzzles using Python and NumPy. By representing the grid as a series of boolean masks, we identify and fill cells that have only one mathematically valid option.

This method proves highly efficient for "Simple" and "Intermediate" puzzles where the solution is deterministic, but still falls short of being able to solve "Expert" level Sudoku puzzles.

3.1 Future Additions

Currently, this algorithm is limited to deterministic solutions. Future development could focus on the following enhancements:

- **Full Vectorization:** Refactoring the `createMasks` function to eliminate one of the `for` loops
- **Backtracking Algorithm:** Implementing a recursive depth-first search (DFS) to handle ambiguous states. This will allow the solver to make ”educated guesses” and backtrack if a contradiction is found, enabling the solution of ”Expert” puzzles.
- **Difficulty Metrics:** Developing a grading system that analyzes the number of passes and branches required to solve a puzzle, categorizing them as Easy, Medium, or Hard.
- **Procedural Generation:** Adding functionality to reverse the solving process to generate valid, unique new puzzles.
- **Generalization:** Abstracting the grid logic to support variations such as 16×16 ”Hexadoku” or irregular ”Jigsaw” Sudoku puzzles.