CS1115 Fall 2013 Project 4 Due Tuesday October 29 at 11pm

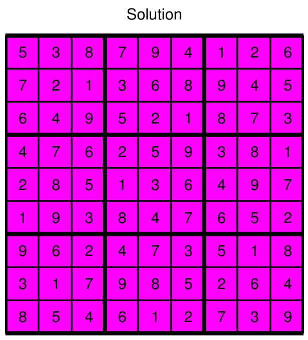
You must work either on your own or with one partner. You may discuss background issues and general solution strategies with others, but the project you submit must be the work of just you (and your partner). If you work with a partner, you and your partner must first register as a group in CMS and then submit your work as a group. Each problem is worth 5 points. One point may be deducted for poor style. No partners are allowed for the Challenge problem.

Objectives

Completing this project will help you learn about two-dimensional arrays, structures, and structure arrays. More on MATLAB graphics and GUI design.

1 Sudoku Generator

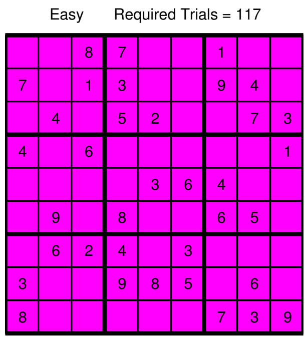
A *Sudoku Matrix* is a 9-by-9 matrix with the property that every row, column, and block has exactly one instance of each positive digit. Here is an example



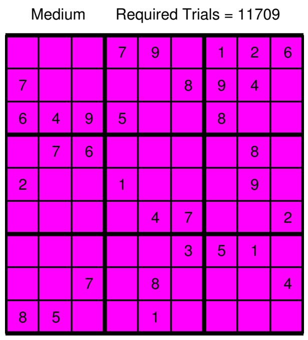
5	3	8	7	9	4	1	2	6
7	2	1	3	6	8	9	4	5
6	4	9	5	2	1	8	7	3
4	7	6	2	5	9	3	8	1
2	8	5	1	3	6	4	9	7
1	9	3	8	4	7	6	5	2
9	6	2	4	7	3	5	1	8
3	1	7	9	8	5	2	6	4
8	5	4	6	1	2	7	3	9

The *blocks* are the highlighted 3-by-3 submatrices. For example, if `S` is a Sudoku matrix, then `S(4:6,7:9)` is the (2,3) block, `S(1:3,1:3)` is the (1,1) block etc.

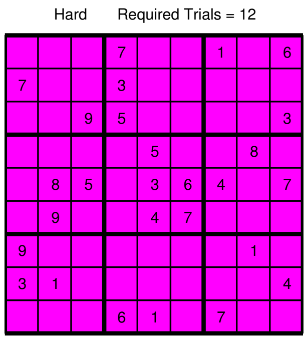
A *Sudoku puzzle* is a Sudoku matrix in which random entries are "punched out" with zeros. Here are three examples with zeros displayed as blanks:


Easy Required Trials = 117

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


Medium Required Trials = 11709

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


Hard Required Trials = 12

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

We make three definitions:

- A Sudoku puzzle is *easy* if it has 36 nonzero entries and each row, column and block has at least 3 positive entries.
- A Sudoku puzzle is *medium* if it has 32 nonzero entries and each row, column and block has at least 3 positive entries.
- A Sudoku puzzle is *hard* if it has 27 nonzero entries and each row, column and block has at least 2 positive entries.

In this problem you write a function that can be used to generate a Sudoku puzzle with prescribed difficulty given a Sudoku matrix. A test function `ShowSudoku` is available on the course website.

Start by implementing a function that can be used to randomly punch out a Sudoku matrix:

```
  function B = PunchOut(A,N)
% A is a 9x9 matrix with positive entries.
% N is a positive integer that satisfies 1<=N<=81
% B is obtained from A by setting 81-N randomly selected entries to zero
```

Some useful MATLAB ideas. For checking out blocks, the `reshape` function is useful. This fragment is instructive:

```
A = [1 2 3; 4 5 6];
v = reshape(A,1,6);      % Same as v = [1 4 2 5 3 6]
w = v(6:-1:1);           % Same as w = [6 5 4 3 2 1]
B = reshape(2,3)         % Same as B = [6 5 4 ; 3 2 1]
```

Suppose `v` is a length-n vector of positive numbers and that we want to set k random entries to zero where $1 \leq k \leq n$. Here is an easy way to do this:

```
[z,idx] = sort(rand(n,1));   % idx will be a length n vector whose entries a permutation of 1:n
i = idx(1:k);                % i will be a length k vector of distinct random integers from [1,n].
v(i) = 0;                    % Set the corresponding entries in v to zero
```

Just because we punch out the "right number" of entries in a Sudoku matrix doesn't mean we have a Sudoku puzzle. For example, `PunchOut(S,32)` doesn't automatically produce an Medium puzzle:

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

It has 32 positive entries, but some of the rows, columns, and blocks have fewer than 3 positive entries violating part of the rule for Medium puzzles. To address this issue, implement the following function

```
  function alfa = CheckOut(A,m)
% A is a 9x9 matrix with nonnegative entries and m is an integer with 0<=m<=9.
% alfa is true if and only if every row, column, and block have at least m positive entries
```

Thus, if `S` has 32 positive entries obtained by punching out a Sudoku matrix, then it is a valid Medium puzzle if `CheckOut(S,3)` is true. MATLAB Hint: if `v` is vector with nonnegative entries, then the value of `sum(v>0)` is the number of positive entries.

To generate a puzzle with prescribed difficulty from a given Sudoku matrix, you can repeatedly punch out the required number of zeros until you get one that "checks out". A while loop can oversee the process. Implement this strategy in

```
  function [T,k] = MakePuzzle(S,Difficulty)
% S is a Sudoku matrix. Difficulty is a string ('Easy', 'Medium', or 'Hard') that specifies the
% difficulty of the puzzle T. k is the number of trials needed to form T.
```

Submit your implementation of `PunchOut`, `CheckOut`, and `MakePuzzle` to CMS.

2 Polynomial Manipulations

Polynomials have a very important role to play in applied mathematics. They are easy to evaluate, differentiate, and integrate. Moreover, they can be used to approximate complicated functions. A very important example are the *Chebychev polynomials*,

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{k+1}(x) &= 2xT_k(x) - T_{k-1}(x) \qquad k = 1, 2, \ldots \end{aligned}$$

This is a recursive definition–the "next" polynomial is $2x$ times the "current" polynomial minus the "one that came before." Notice that T_k has degree k.

Other examples are the *Legendre polynomials*

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ P_{k+1}(x) &= \frac{2k+1}{k+1}xP_k(x) - \frac{k}{k+1}P_{k-1}(x) \qquad k = 1, 2, \ldots \end{aligned}$$

and the *Hermite polynomials*

$$\begin{aligned} H_0(x) &= 1 \\ H_1(x) &= x \\ H_{k+1}(x) &= xH_k(x) - H'_k(x) \qquad k = 1, 2, \ldots \end{aligned}$$

In this problem you write various functions so that the given test script plots examples of these polynomials and displays their coefficients.

Start by downloading `ShowPoly` and note that it contains a "Make" function that defines a structure that can be used to encode a polynomial:

```
   function P = MakePoly(a)
% a is a length n vector.
% P is a two-field structure that encodes a polynomial p(x).
% P.a specifies the coefficients:
%   p(x) = P.a(1) + P.a(2)x + a(3)x^2 + ... + P.a(n)x^(n-1)
% P.deg is the degree of the polynomial (i.e.,P.deg == n-1)
deg = length(a)-1;
P = struct('a',a,'deg',deg);
```

For `ShowPoly` to work, you must implement a number of subfunctions. Briefly

`function PlotPoly(P,L,R)`	`plots P across [L,R]`
`function yVals = EvalPoly(P,xVals)`	`evaluates P at the xVals(1),...,xVals(n)`
`function R = ScalePoly(alfa,P)`	`scales P by scalar alfa to get polynomial R`
`function R = DerPoly(P)`	`R(x) is the derivative of P(x)`
`function R = SubPoly(P,Q)`	`R(x) is the polynomial P(x) - Q(x)`
`function R = MultPoly(P,Q)`	`R(x) = P(x)*Q(x)`
`function T = MakeChebychev(n)`	`T(k) is the kth Chebychev polynomial, k=1:n`
`function P = MakeLegendre(n)`	`P(k) is the kth Legendre polynomial, k=1:n`
`function H = MakeHermite(n)`	`H(k) is the kth Hermite Polynomial, k=1:n`

Complete specifications are given in`ShowPoly`. Your implementations should make effective use of the given polynomial structure. Submit your fully developed implementation of `ShowPoly` to CMS.

3 IsingGUI

An interesting two-dimensional Monte Carlo simulation that physicists use to understand pole alignment in a magnetic substance involves the *Ising model.* The components of an Ising model are an n-by-n array A of *cells* that have one of two states (+1) and (-1), a probability p, and a temperature T. During the simulation, the states of the cells change in a probabilistic fashion. Whether or not a particular cell changes state depends upon the temperature and the states of the four neighbor cells.

The simulation is initialized by setting up an n-by-n matrix A where a_{ij} is $+1$ with probability p and -1 with probability $1-p$.

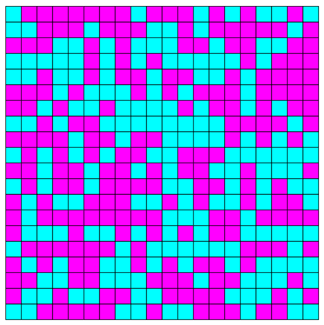
To describe how the cells change state we need to define the notion of *potential.* The potential of cell (i,j) is the value of `A(i,j)*(N+E+S+W)` where `N` is the state of the north neighbor, `E` is the state of the east neighbor, `S` is the state of the south neighbor, and `W` is the state of the west neighbor. In the Ising setting,

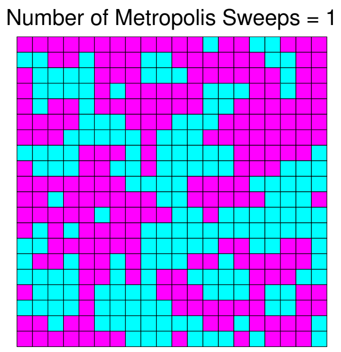
The east neighbor of a cell (i,n) on the east edge is cell $(i,1)$.
The south neighbor of a cell (n,j) on the south edge is cell $(1,j)$.
The west neighbor of a cell $(i,1)$ on the west edge is cell (i,n).
The north neighbor of a cell $(1,j)$ on the north edge is cell (n,j).

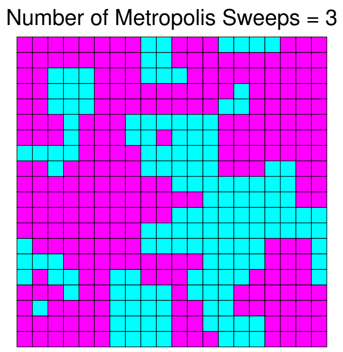
Here is what happens at each "time step" in the simulation:

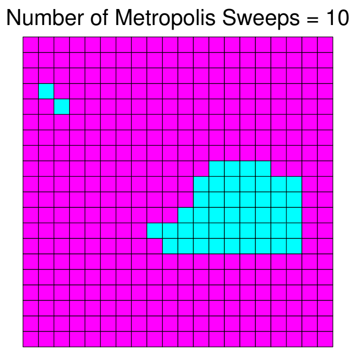
- A cell is chosen at random. That is, we choose random integers i and j that satisfy $1 \le i \le n$ and $1 \le j \le n$.
- Let P be the potential of the (i,j) cell. If $P < 0$, then the cell's state is always flipped. If $P \ge 0$, then the cell's state is flipped with probability $e^{-2P/T}$. Here T is a given temperature.

A *Metropolis sweep* consists of n^2 time steps. Here are some snaphots of an $n = 20$ simulation with cyan encoding the $+1$ state and magenta encoding the -1 state:









The Ising model captures the idea that subject to random fluctuations, a cell will tend to have the same state as its neighbors, i.e., its magnetic polarity tends to be that of its neighbors. You are to complete the given GUI template `IsingGUI.fig` so that it supports handy experimentation with the Ising model. Requirements:

- A color-coded version of the current array is to be displayed in the graphics window at all times.
- The "New pushbutton" is to generate a new initial array using the values associated with the n-slider and the p-slider.
- Any change in the value of n, p, or T should produce a new initial array.
- The "Start pushbutton" must initiate a simulation that steps through 10 Metropolis sweeps. During the simulation, the displayed array must be updated after each time step. (You may want to do a smaller number of Metropolis swweps during debugging.)

It is handy for the A-matrix that encodes the states to be a `handles` variable. Given `handles.A`, the color-coded display can be generated. However, that requires n^2 `fill` commands. Clearly, after each time step we have to (possibly) update the color in just one tile. To incorporate this idea, you need to use this "handles graphic" idea:

```
% set up H...
hold on
H(1,1) = fill([0 1 1 0],[0 0 1 1],[1 0 0])    % A red unit square
H(2,1) = fill([0 2 2 0],[0 0 1 1],[1 0 0])    % A green unit square
H(1,2) = fill([0 1 1 0],[1 1 2 2],[1 0 0])    % A blue unit square
H(2,2) = fill([0 2 2 0],[1 1 2 2],[1 0 0])    % A black unit square
hold off
% Update the (2,1) tile color...
set(H(2,1),'facecolor',[1 0 1])               % The (2,1) tile is now magenta
```

Play with this! `H(1:2,1:2)` is a matrix of handles. In your problem, you can have an n-by-n array of handles `handles.H`. A command like

```
 set(handles.H(i,j),'facecolor',[[0 1 1])
```

can be used to redisplay tile (i, j) in cyan.

You may want to code up and get clear on the initialization and simulation before you start integrating that stuff into the GUI template. Submit you final `IsingGUI.fig` and `Ising.m` to CMS.

Challenge 4: Roman Numeral Checker

This problem is about the "grammar" of the Roman Numeral "language." Start by reading FVL §9.2 to get familiar with Roman Numerals. You are to implement a function

```
  function x = RomNumVal(s)
% s is a string. If it encodes a valid Roman numeral then x is its numerical value.
% Otherwise x returns 0.
```

You are NOT to use the strategy that is laid out in §9.2. Instead, you are to check that these rules are satisfied.

Rule 1. `s` must have positive length and consist only the characters `I`, `V`, `X`, `L`, `C`, `D`, `M`.

Rule 2. The characters `V`, `L` and `D` can occur at most once.

Rule 3. The characters `I`, `X`, `C`, and `M` can occur at most three times.

Rule 4. The only valid length-2 substrings where the value of the second numeral is greater than the value of the first numeral are `'IV'`, `'IX'`, `'XL'`, `'CD'`, `'CM'`. When this happens, the value of the first numeral is negated.

If these four rules hold for `s`, then you can compute the value of `s` by summing the (possibly negative) values of each character. Recall that `'I'`=1, `'V'`=5, `'X'`=10, `'L'`=50, `'C'`=100, `'D'`=500, `'M'`=1000.

Is it possible for `s` to satisfy the four rules without encoding a valid Roman numeral? If so, give an example. Submit `RomNumVal` to `CMS`. Points may be deducted if your rule checking is cumbersome and/or insufficiently commented.