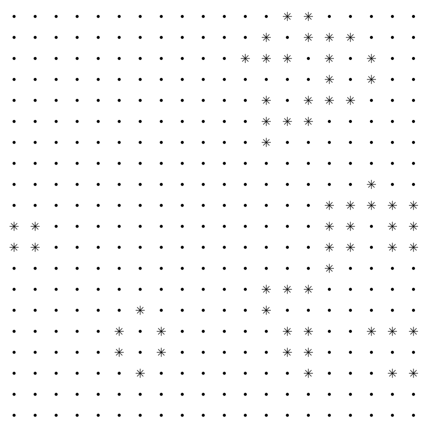
# CS1112 Fall 2022 Project 4 due Thursday 10/27 at 11pm

You must work either on your own or with one partner. If you work with a partner you must first register as a group in CMS and then submit your work as a group. *Adhere to the Code of Academic Integrity.* For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is certainly not OK to copy code from another person or from published/Internet sources. If you feel that you cannot complete the assignment on you own, seek help from the course staff.

## Objectives

Completing this project will solidify your understanding of 2-dimensional and 3-dimensional arrays. In problem 1, you will write a set of functions to simulate Conway's Game of Life. In problem 2, you will use MATLAB as a tool to solve a system of linear equations. In problem 3, you will work with digital images, the type `uint8`, and MATLAB graphics. Pay attention to the difference between `uint8` and MATLAB'S default type `double`. Problem 3 is now posted at the end of this document.

## 1 Conway's Game of Life



Conway's "Game" of Life is not actually a game that one plays; rather it is a simulation from some initial state following a set of rules. The state of the game is a grid (matrix) of elements, sometimes called cells. A cell can be either alive or dead. In one step of the game, the cells will either live or die according to some rules. You will write three functions to simulate this game. The rules stated below are taken from the Wikipedia entry on Conway's Game of Life.

### 1.1 One step

Implement the following function as specified:

```
function outM = oneSweep(inM)
% One step of Conway's Game of Life.
% inM and outM are 0-1 matrices of the same size.  1 is live; 0 is dead.
% inM is the initial state; outM is the state after 1 step of the game.
% Every "cell" in the matrix interacts with its immediate neighbors, which
% are the cells that are horizontally, vertically, or diagonally adjacent.
% At this time step, the following transitions occur:
%   A live cell with fewer than 2 live neighbors dies (under-population).
%   A live cell with 2 or 3 live neighbors lives on.
%   A live cell with more than 3 live neighbors dies (overcrowding).
%   A dead cell with exactly 3 live neighbors becomes live (reproduction).
```

One step of the game involves visiting every cell and determining its new state. Be sure that for every cell you use the initial state when computing the transition. While the interior cells of the matrix each has exactly eight neighbors, the cells in the first and last rows and first and last columns each has fewer than eight neighbors.

#### Helpful syntax and function

```
m = [ 2 4 2 6; ...
      0 1 7 4; ...
      9 3 5 8];
x = m(2:3,1:3);  % x is 2-by-3 matrix containing [0 1 7; 9 3 5]
y = sum(x);      % y is row vector storing column sums of x: [9 4 12]
z = sum(y);      % z is sum of vector y, same as the sum of matrix x: 25
```

## 1.2 Draw the state of the game

Implement the following function as specified:

```
function drawState(m, s)
% m is a 0-1 matrix.  s is the step number in the game.
% Draw an asterisk at (c,-r) if m(r,c) is 1 (live).
% Draw a dot at (c,-r) if m(r,c) is 0 (dead).
% Display the step number in the title area of the figure.
```

In order to draw the state of the Game of Life grid, we need to relate row and column numbers of matrix `m` to x and y coordinates. The *column* number maps to the x-coordinate, but notice that row numbers increase going down while the y-coordinate increases going up, which is why we map row `r` to the y-coordinate `-r`.

Begin with the commands `cla` for clearing (refreshing) the figure axis area and `hold on`. End with the command `hold off`. Choose any color combination you like and you can experiment with different marker and font sizes. See Appendices A.6 for font size and A.10 for marker formats in *Insight*. (Or use the MATLAB documentation.)

## 1.3 The game, finally

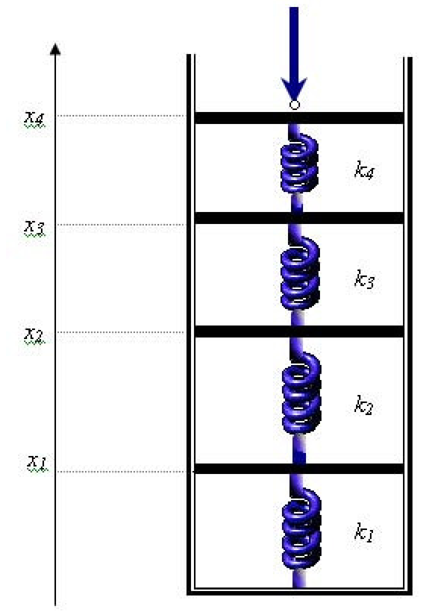
Implement the following function as specified:

```
function m = gameOfLife(m, n)
% Simulate n steps of the Game of Life.
% Pre:  m is a 0-1 matrix representing the initial state.  m is not empty.
% Post: m is a 0-1 matrix representing the state after n steps of the game.
```

In this function you will call your functions `drawState` and `oneSweep` repeatedly in order to simulate the game. Below is a skeleton of the function showing and explaining the relevant graphics commands.

```
close all
figure
axis equal off

% Call drawState to show the initial state

% Simulate n steps
for k= 1:n
    pause(.1)  % During program development, change to pause() so that you
               % can manually advance the simulation and check each step

    drawnow  % Force MATLAB to complete drawing in the figure window before
             % moving on.  Useful in animation (in loop) when the computation
             % gets done faster than drawing on the screen.

    % Write your code to simulate one step and draw the state

end
```

## 2 System of Linear Equations



Systems of linear equations often arise from the modeling of systems of interconnected elements. In your physics class you may have studied the displacement that results from applying a force to a set of blocks connected by springs. Such a model gives a coupled set of equations that must be solved simultaneously; the equations are coupled because the individual parts of the system are influenced by other parts.

*[Our objective for this problem is to show how you can use MATLAB as a solver of systems of linear equations. We do not expect you to know linear algebra nor are we trying to teach linear algebra. We will use the solver as a "black box," so your only real task is to turn a set of given equations into a matrix and a vector and then use the solver, which is just an operator. So there's no need to be afraid of the physics or math! This discussion and problem is adapted from Introduction to Computing for Engineers by Chapra and Canale.]*

Consider a set of $n$ linear algebraic equations of the general form

$$\begin{array}{cccccc} a_{11}x_1+ & a_{12}x_2+ & \ldots+ & a_{1n}x_n = & b_1 \\ a_{21}x_1+ & a_{22}x_2+ & \ldots+ & a_{2n}x_n = & b_2 \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ \cdot & \cdot & & \cdot & \cdot \\ a_{n1}x_1+ & a_{n2}x_2+ & \ldots+ & a_{nn}x_n = & b_n \end{array} \tag{1}$$

where the $a$'s are known constant coefficients, the $b$'s are known constants, and the $n$ unknowns, $x_1$, $x_2$, ..., $x_n$, are raised to the first power. This system of equations can be expressed in matrix notation as

$$\mathbf{Ax} = \mathbf{b}$$

or

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ \cdot \\ \cdot \\ b_n \end{bmatrix} \tag{2}$$

To solve this linear system of equations is to find the values $x_1$, $x_2$, ..., $x_n$ such that $\mathbf{Ax} = \mathbf{b}$. In MATLAB, the solution can be found using the backslash, called the *matrix left divide* operator:

```
x = A\b
```
(3)

where `A` is the $n$-by-$n$ matrix of coefficients, `b` is the length $n$ *column* vector of constants, and the result `x` is the length $n$ column vector of values such that $\mathbf{Ax} = \mathbf{b}$.

**Your job:** Write a script `springEqns.m` to find the positions $x_1$, $x_2$, $x_3$, and $x_4$ of the spring-mass system shown above when a force $F$ of 2000 lbs is applied. The spring constants $k_1$ through $k_4$ are 100, 50, 75, and 200 lb/in, respectively. The force-balance equations that define the relationships among the springs are

$$k_2(x_2 - x_1) = k_1x_1 \tag{4}$$
$$k_3(x_3 - x_2) = k_2(x_2 - x_1) \tag{5}$$
$$k_4(x_4 - x_3) = k_3(x_3 - x_2) \tag{6}$$
$$F = k_4(x_4 - x_3) \tag{7}$$

You need to do the following:

1. Rewrite these equations in the general form of (1) by collecting terms together, where the unknowns are $x_1$, $x_2$, $x_3$, and $x_4$. This means for each equation, re-arrange it so that the unknowns are on the left hand side while the constants $F$ (or $-F$) and 0 are on the right hand side. For example, (4) can be rearranged as follows:

$$\begin{aligned} k_2(x_2 - x_1) &= k_1 x_1 && \text{Original} \\ k_2(x_2 - x_1) - k_1 x_1 &= 0 && \text{All unknowns moved to left side} \\ (-k_1 - k_2)x_1 + k_2 x_2 + 0x_3 + 0x_4 &= 0 && \text{Re-order left side by the 4 unknowns} \end{aligned}$$

   The final rearranged equation should have all *four* unknowns on the left side; some of their coefficients may be zero, as shown above. Do this rearrangement of each equation ((4)-(7)) by hand (on paper)—you do not need to include this step in your script or submit it.

2. In your script, create the 4-by-4 matrix `A`—the coefficients of the 4 unknowns in 4 equations—and the length 4 column vector `b`—the constants on the right side of the re-arranged equations—as shown in (2).

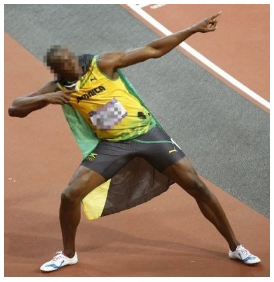
3. Apply the matrix left division operator as shown in (3) to solve for $x_1$, $x_2$, $x_3$, and $x_4$.

4. Finally display the values of $x_1$ through $x_4$ neatly.

Note that since $k_1$, $k_2$, $k_3$, $k_4$, and $F$ are important parameters, you should store their values in variables and use those variables when creating the necessary matrices and/or vectors.

## 3 Pixelation

"Fastest Man on Earth"


Face and name pixelated
(Pose gives him away!)

Often seen online, photos will have people's faces, license plates, etc. blurred out to protect people's privacy. In this problem, you will perform a similar operation. You will implement a set of MATLAB functions to *pixelate* user-selected parts of a digital image. Here are the main steps: (1) Allow a user to select an area of an image that they wish to pixelate. (2) For the selected area, divide the area into non-overlapping square blocks of pixels. (3) Calculate the average RGB intensities for each block. (4) Draw a square filled with the average color for each block over the image. These steps repeat until the user chooses not to select another area to pixelate.

Included in `P4_Files.zip` are two function files that you need to complete, (`multiSelect.m` and `pixelate.m`), a completed function that you will use (`DrawRectNoBorder`), and the png image file that we used for demonstration (`usainBolt.png`). Feel free to use your own images for testing. Be sure to check out the video example (`PixelationExample`) of how our pixelation process in this project should execute.

**Workflow**. The function `multiSelect` will allow a user to select which area of an image to pixelate while the function `pixelate` carries out the pixelation. The beauty of using functions is that you can work on the tasks independently, and here you can choose which function to work on first! Read the specifications of both functions and this document. Then decide for yourself which function to work on first.

### 3.1 Select multiple areas

**Read the specifications of the function `multiSelect` in the file `multiSelect.m`.** Below we give some additional information on the function; read it and then implement the function. Be careful that this function deals with user interaction only and calls another function, which you will implement later—to do the pixelation.

**Bounding box.** Two points on the Cartesian plane, assuming that they don't share the same x or the same y coordinate, can be used as the opposing corners of a rectangle with sides that are parallel (perpendicular) to the axes. A user can select two opposing corners in any order (top left first then bottom right, or bottom left first then top right, or . . . ). Your code should be able to identify the rectangular region that the user selects no matter which two opposing corners the user clicks. You do not need to be concerned in this function about two user-selected points that share the same x or y coordinate or a bounding box that is too small; the next function will handle those scenarios. **Take a look at the specification of function `pixelation` now so that you know what values you need to compute in `multiSelect` in order to later call `pixelation`.**

**Row and column indices vs. x and y coordinates.** The statement `[xi,yi]=ginput(1)` accepts one user mouseclick on the current figure window and returns the x- and y-coordinates of the click in `xi` and `yi`, respectively. In a typical figure window with Cartesian axes, x values increase to the right while y values increase going up. However, image data is stored in an array such that the top row of pixels of the image corresponds to row number 1, the row below that is row number 2, and so forth. Notice that the row number increases going down an image. The `imshow` and `image` function flips the direction of the y-axis to match the image convention.

When we use built-in function ginput() to obtain the location of a mouseclick in a figure window displaying *image* array data, the "x-coordinate" returned corresponds to the column number of the pixel clicked while the "y-coordinate" returned corresponds to the row number of the pixel clicked. If you compare the returned "y-coordinates" of two clicks made one below the other on an image, you will see that the lower clicked location has the larger row number. Meanwhile, the column numbers increase to the right, like the x-axis. Note the difference in order: when we use coordinates we (usually) say x first and then y, but when you access an element in an array you specify row (corresponding to y) first and then column (corresponding to x). Be careful not to mix them up in your code—use meaningful names for your variables so that it's crystal clear whether a variable refers to a coordinate or an index! Finally, you must round a coordinate before you can use it as an index of an array.[1]

## 3.2 Pixelate an area

**Read the specifications of the function `pixelate` in the file `pixelate.m`.** Below we give some additional information and hints on doing the pixelation; read them and then implement the function. We encourage you to design and implement subfunctions in order to modularize your code and facilitate testing, but it is not a requirement in this problem.

**Pixelation as a set of blocks.** As you can see from the example, the pixelation is a set of colored squares drawn on top of the image. The squares are drawn adjacent to one another within a rectangular area—the pixelation area. Each square has the color that is calculated (averaged) from the block of pixels "underneath." As stated in the specifications, if the pixelation area (selected by the user) is too small or too narrow, no pixelation will be done and execution of the program should be stopped with an error message (explained below).

**Trim and block.** When necessary, your function should trim the selected area to be pixelated so that its number of rows and column of pixels are each a multiple of `n`, starting from the top left corner (crop from the right and bottom). For example, if a 3D `uint8` array `ex` has the size $159 \times 84 \times 3$ and `n` is 10, then keep `ex(1:150,1:80,1:3)`, which results in 15 rows by 8 columns of blocks of pixels where each block has 10-by-10 pixels.

**Halting execution with an error message.** You've seen before that MATLAB shows an error message in red and gives an audible "ding" when it encounters a run-time error. We can write our own code to cause program execution to stop under some condition and provide our own, informative, error message. The built-in function to use is `error`; here is an example unrelated to this project that makes use of the function `error`:

```
x = input('Enter a positive value: ');
if x<=0
   error('You must enter a positive value!')
end
```

If you execute the above code fragment and enter -3 when prompted, then the program stops, dings, and displays in red in the Command Window the message "You must enter a positive value!"

**Types `uint8` and `double`.** The image data (`A`) that we get from a png file have the type `uint8`, but when we use MATLAB graphics later (to draw the averaged colored blocks) we need the RGB values as a `double` in the range of 0 to 1 where [0 0 0] represents black and [1 1 1] represents white. Notice that the array `colr` in the function should be in type `double`, which means that you need to scale an integer in the range 0 to 255 (remember that `uint8` variables only store integers between 0 and 255, inclusive) to a real value in the interval [0,1].

**Averaging.** For each block of pixels, you calculate and store one average red intensity, one average green intensity, and one average blue intensity; the average is done over $\texttt{n}^2$ pixels.

**Testing.** Check now that your code produces the correct number of blocks and that the average color values are correct. You can use the given image data, of course, and you can also create smaller, simpler data sets for testing. Consider the following examples:

```
bolt = imread('usainBolt.png');   % bolt is the 3d uint8 array of an actual image
data = uint8(rand(11,14,3)*255);  % data is a small generated 3d uint8 array for testing
u1 = pixelate(data,1,14,1,10,2);       % Test1: no cropping. u1 should be 5 by 7 by 3 array
u2 = pixelate(bolt,21,75,121,180,10); % Test2: crop columns but not rows. u2 should be 6 by 5 by 3
                                       % DO MORE TESTS!
```

[1]It may look as though the points that you select using `ginput` have integer valued coordinates, but any user or MATLAB (automatic) action that re-sizes the figure window showing the image could change the returned values from `ginput` to type `double` values. Therefore be sure to round a coordinate to an integer value if you intend to use it as an index.

Choose test cases to cover the different actions that your code needs to perform: crop rows, crop columns, crop both rows and columns, crop neither rows nor columns, and stop program with error message. Check that the number of rows of blocks, the number of columns of blocks, and the dimensions of the returned array are correct. Then confirm "by hand" that the averaged color values in the returned array (check several values) are correct. You can see why you want to create "small" test cases! Finally, we mention here that the returned array from this function should not need to be used in function `multiSelect` later. `colr` is specified for this function for ease in independent testing by you now and by the course staff later!

**Drawing the blocks.** The effect of pixelation can be achieved in multiple ways. In this problem we ask you to use MATLAB graphics to draw colored squares on top of the image to *display* the effect of pixelation without modifying image data.[2] To draw the squares, use the given function `DrawRectNoBorder`, which is a modified version of the function `DrawRect` that you have used before in past projects. `DrawRectNoBorder` differs from `DrawRect` in that it omits the outline of the rectangle (while still filling the rectangle with the specified color). Where do you draw the squares? You can calculate the x and y coordinates that you need using `lc`, `tr`, and `n` (the inputs to the `pixelate` function). Recall from the discussion for `multiSelect` that you can convert between column and row indices and x and y coordinates.

Submit all files on CMS. For problem 1, that is `oneSweep.m`, `drawState.m`, and `gameOfLife.m`. For problem 2, that is only `springEqns.m`. For problem 3, that is `multiSelect.m` and `pixelate.m` on CMS.

[2]An alternative way to do pixelation is to overwrite the RGB data of the image with the averaged values in the blocks of the pixelation area, modifying the data. Do not do this for our problem in this project.