***Problem 2: Fuzzy Matrix Problem***

For the exact solution, use a non-linear least squares solver (MATLab function, lsqnonlin). The function uses a trust-region-reflective algorithm, which can only work when *b* is at least the size of *x*. The second method used by the MATLab function does not accept bounds on generated solutions, so there would be no guarantee that the generated solution would be fuzzy. The trust-region-reflective algorithm uses the concept of trust-regions to approximate the function being optimized. A simpler function is generated that accurately represents the original function in a small region, referred to as the trust-region. The algorithm then minimizes the approximation function over the trust-region. There could potentially be issues with this method, as the least-squares function, ||*A*◦**x** – *b*||2, can be non-continuous.

My proposed soft-computing approximate solution method is a genetic algorithm with pre-processing to reduce the potential *xi* values to sets of discrete cases, *Si*, determined by values in *A* and *b****.*** The set of possible values for each *xi* can be updated after some set number of iterations if the values present in the population have converged to some smaller sub-set of values. This will allow the algorithm to explore various resolutions of solutions. The encoding for each solution will be a vector of integers *X*, where each entry corresponds to a particular member of *Si* such that *xi* ∈ {1, |*Si*|}.***Problem 3: Max Cut***

A simple approach to solving the Max Cut problem exactly involves encoding the solution as **x** = {0,1}n, where *n* is the number of vertices in the graph, and each entry represents whether the corresponding vertex is in one partition or the other, as the max cut problem partitions the graph into two groups. Since **x** has only binary values, and since each solution vector has a complement solution vector for which the sum of the solutions is 2n, the size of the solution space is 2n-1, so any intuitive algorithm to determine the solution runs in super-polynomial time with complexity *O(2n-1)*. The graph cut decision problem is in *NP*, but it would not seem that checking whether a proposed cut is the max cut cannot be checked in polynomial time unless the max cut problem is in *P*.

A Tabu-based search is an effective soft-computing method to find the max cut of a graph. The principle idea of Tabu search is that decisions made in the recent past are not repeated or undone. This can force an algorithm to switch to proposed solutions that are worse than a previously found solution, allowing the algorithm to burrow out of local minima. To prevent permanently moving away from an optimal solution, the best solution found thus far is stored in memory.

For this problem, the algorithm moves through solutions by considering a subset of neighbors, *S*, for the current solution **C**, and computes the quality score for each of these neighbors. If the neighbor with the best quality score was not obtained by a move in the Tabu list, *T*, then it becomes the new current solution, **C’**, regardless of whether its quality score is better or worse than the starting proposed solution for the current iteration. If the best neighbor is arrived at via a move in *T*, then it becomes **C’** only if its quality score is strictly greater than the best quality score found so far by the algorithm (aspiration criteria). If this is not the case, then all neighbors in *S* that are obtained by a move in *T* are removed from consideration, and the remaining best neighbor is selected as **C’**. Various checks are in place to deal with situations where all neighbors in *S* correspond to moves in *T*. In any case, the move corresponding to the transition from **C** to **C’** is placed in *T* for the pre-set tenure period; if it was already in *T*, than its remaining time in *T* is reset to the tenure period.

Encoding solutions for the Tabu algorithm is the same as for the exact case, a binary vector of length *n*. The neighbors of **C** are then defined as all vectors *N* that can be obtained by flipping a single bit in **C**. Therefore, each **C** has a rather small number of neighbors, *n*, in comparison to the size of the search space. Using a single flip as a neighbor instead of multiple flips makes encoding and transitioning rather simple, while also minimizing the number of neighbors for a given solution, thereby minimizing the potential number of solutions to consider at each iteration. The initial solution is created pseudo-randomly using the MATLab ‘rand’ function. Modifications in creating this starting solution could be made if something was known about the adjacency matrix for the graph being analyzed.