**Structure of the code:**

1. optimize() is the top level function for getting the final optimized results.
2. inputScene() is where all the inputs in the scene is declared and described.
3. init () initializes the scene and calculates a few important things like the structure of the decision variables and their respective range.
4. constructSPM() generates the coefficients from the equations. The equations are constructed one by one in a self explanatory manner.
5. Functions generateIndices(), conditionalIndices(), resolvePos(), rangeVarCoeff(), doubleVarCoeff() are utility functions in this framework which is heavily used to construct the equations easily without writing endless nested loops for each one of them. Every one of these are explained in the following sections.

Any new set of equations can also be constructed with these.

**1. function Optimise() in optimise.m**

[ u, v, w, wR, x0, x, xR, y0, y, yR, z] = optimize (optional\_mat\_filename)

**Input:** optional . If the input is left blank, it will automatically take inputs form the inputScene.m file. If there is a mat file saved with a different set of inputs, provide the filename here.

**Output:** 1. Series of decision variables. 2. Generates the coefficient matrices and saves them in CPLEX lp format as “diaster1.lp” name can be easily changed in the code.

Call structure (Algorithm):

Optimise() ->

Step 1) if (output empty) call inputScene()

Step 2) call Init()

Step 3) call [f, A, LHS, RHS, LB, B] = constructSPM() [this step constructs the coefficient matrices in a sparse matrix format]

Step 4) save the data in CPlex lp format.

Step 5) call X = cplex.solve() [CPLEX optimizer]

Step 6) Re-Construct all the decision variables form the output X.

**2. function InputScene() in file inputScene.m**

inputScene is the place where you input the data about any particular scene. The entire code is heavily commented and extremely self explanatory. There are three things to know.

1. The variable *tick* is used toconvert the time from minutes to discrete time ticks. All the timing inputs shall be in minutes. Specifying how many minutes is wanted in one tick will calculate times in ticks instead of minutes.
2. It is assumed that there are 3 simultaneous disasters of the same type are happening in 3 cities. The L stores the disasters and for 8 cities, 8C3 + 8C3 = 112 different combinations of disasters are there. This makes S = 112.
3. It is assumed, every s in S has same chance of occurring and the probability is calculated automatically 1/S.

**3. function init() in file init.m**

The init() function defines the structure of the decision variables (dVar) . It is heavily commented too and it was important to set the order of the dVars which is set alphabetically. The 6 possible indices also have a certain order [S, L, R, I, J, T]. The dVarRange variable calculates the range of each dVars to convert them all into a single very long vector. dVarRange is very useful to find the location of a particular decision variables in the long decision variable vector needed for CPLEX.

Index Var Order: s, l, r, i, j, t

Decision Var Order: u, v, w, wbar, x0, x, xbar, y0, y, ybar, z,

**4. function constructSPM() in constructSPM.m**

constructs the equation to generate the coefficient matrices from each equation needed for the CPLEX optimizer.

The equations are constructed one by one and all of them are in separate numbered functions. These are all self explanatory. There are two types of equations. Equalities, i.e. RHS = 0 and inequalities, i.e. RHS <=,>= (some values)

Construct equations till Eq 15 are equalities and they return only the coeff matrix *spm*.

Equations from 16 to end are inequalities and they return coeff matrix ‘*spm’* and RHS vector ‘*bspm’*. The LHS and RHS are then constructed accordingly (whether it is <=, or >=)

**Construction Framework:**

Construction of the equations are done by a fleet of 5 functions in the framework which makes it much easier to construct any equation. The five functions that let this while task happen are:

generateIndices(), conditionalIndices(), resolvePos(), rangeVarCoeff(), doubleVarCoeff()

**The central idea of the construction framework:**

The central idea is to generate the relevant combinations of the index variables S, L, R, I, J, T as needed for a particular decision variable. This will give a collection of these index variable combinations for each iteration of the particular decision variable.

For example, let’s consider the decision variable U

U has 4 indices -> S, L, R, T

Let’s say N = S\*L\*R\*T

The idea is to generate N different combinations of the index variables for U. The index Matrix will look like :

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

[1, 1, 1, 0, 0, 2]

.

.

[3, 1, 2, 0, 0, 4]

.. so on.

NOTE: As the I, and J index are not needed fro U, these have value 0 throughout.

With this (N x 6) matrix of possible index, it is much easier to pick one row of indices and try to find the position of the particular decision variable in the long decision variable vector.

So suppose the full vector has 1000000 variable elements. The job is to find position of U (s,l,r,0,0,t) in the full vector. Once the position is found, the position is added with the necessary coeff value (1, -1, or the appropriate coefficient)

**Framework Functions:**

**Index generation :** generateIndices(), conditionalIndices()

**Finding the position of a dVar:** resolvePos()

**Additional Utilities:** generateIndices(), conditionalIndices()

**5.1 function generateIndices() in generateIndices.m**

generates the index combinations where the indices runs from 1 to their maximum values

call structure: indexArray = generateIndices( [maxS, MaxL, 0, maxI, 0, maxT], Tstart)

Input: 2 inputs. 1st input is a vector consist of the max values of the index combination wanted., second input is the start point of T because in some cases T might start later than 1. Usually it is always 1.

Output: This generates the Nx6 matrix of index combinations described before.

**5.2 function conditionalIndices in conditionalIndices.m**

Exactly same as generateIndices but generates them conditionally where there are dependencies such as L depends on S, R depends on L, I depends on L, J depends on I and T depends on S.

call structure: indexArray = conditionalIndices( [1, 1, 0, 1, 1, 1], Tstart)

Input: 2 inputs. 1st input is a vector consist of the index combination wanted. NoTE: No need to give max values because the values are conditional. Second input is the start point of T because in some cases T might start later than 1. Usually it is always 1.

Output: This generates the Nx6 matrix of index combinations described before.

**5.3. function resolvePos in resolvePos.m**

resolves the position of a decision variable inside the large decision variable vector.

**call structure:** position = resolvePos( dVar, [s, l, r, i, j, t])

**Input:** 2 inputs.

1) dVar: The decision variable number according to its order mentioned before. (u = 1, v= 2, .. x = 6, xBar = 7 .. etc. )

2) the index of dVar as the 6 element vector

**output:** position

NOTE: ERROR CODE of position. If there is an error resolving the position, it’ll output a negative value. This is a great way to check whether the position is correct.

**5.4. function rangeVarCoeff()**

Generates a range of decision variable coefficients at once. For example, the equation 10.a, 10.b, 10.c is the prime example of it.

**Call structure :** [coeff\_matrix] = rangeVarCoeff(dVar, [indicesRange], tStart, coeff)

**Input:** usual inputs. dVar, the indices range, tStart = 1 and coeff is the coefficient value for each of them.

**Output:** full coefficient matrix for the whole range of decision variable.

5.5. doubleVarCoeff()

Sometimes there are two nearly adjacent values of a single decision variable.. such as Y(t) and Y(t+1). This function outputs two different positions for these kind of pairs.

**Call structure:** [p1, p2] = doubleVarCoeff(dVar, [index Array] )

**Input:** Self explanatory.

**Output:** 2 outputs for position 1 and 2 for the two vars in the pair.

Discrepancy in data:

**Location and Preferences:**

Table 2 says:

City adjacent cities in order of preference

C D B G H **C** E F

D C B E G H **D** F

The C and D should not be there. In both cases A is missing and A should be there. It was pointed out in an earlier report but was denied. This time, the data provided in the mat file to me was consistent with my finding. In the provided data to me, the cell L{1,3} has 3 rows and 8 columns for the location. 1st row is for cities (1-8 starting form 2nd column) Third row is for preferences. Preferences for city C (3) is 6 3 2 7 8 4 5. This is consistent with my finding that there’s indeed a mistake in the provided scenario description.

Moreover, Moving on to the location data for city D ( L{1,4} ), we have another discrepancy in the mat file data which I think is a mistake. The preference row says: 7,3,2,4,8,6,6  
The preference row again is consistent with my finding that it should be A, not D. However, there is two same preferences (6) for two last cities G and H. This according to me should have been 5,6.

**Location and response Time:**

According to the scenario given to me, I found some of the response times are slightly different in the given data. I’ve checked my times meticulously and there are no errors in it. My raw data can be seen as minutes which I divided by variable tick. In default tick=20. It can be later changed to any value.

For example, from city E to H, it takes 380 mins. Which is 19 ticks (380/20). However in the data provided it is given as 20 ticks. I guess it was calculated erroneously along HCDE route which is 400 mins = 20 ticks.