Here we present a short documentation with the description of the main functionalities.

This code is responsible for the results in the respective paper, which concerns, in summary, different situations in which agents learn independently to play a spatial prisoner's dilemma in an environment with defects, each with their own Q-table, then composing a multi-agent reinforcement learning framework, but with single agent characteristics.

Variables and constants

- **NUM_CONF**, **LSIZE** → number of independent samples and lattice size.
- **INITIAL_STATE** → defines the initialization pattern, which is then passed to the switch statement:

```
switch (INITIALSTATE)
  {
       case 1 : fprintf(freq,"Random\n");
           break;
       case 2 : fprintf(freq,"One D\n");
           break;
       case 3 : fprintf(freq,"D-Block\n");
           break;
       case 4 : fprintf(freq,"Exact\n");
           break;
       case 5 : fprintf(freq,"2 C's\n");
           break;
  }
```

- **PROB_C**, **PROB_D** → probability of initialization as C or D.
- **TOTALSTEPS**, **MEASURES** → total Monte Carlo steps and total measures to save in file.
- **NUM_NEIGH** → number of neighbours.
- **FRANDOM1** → variable to generate a random number uniformly between 0 and 1 using gsl.
- **C**, **D**, **MOVE**, **Cindex**, **Dindex**, **MOVEindex** → integers associated with the actions and their respective indexes
- **STATES[NUM_STATES]**, **ACTIONS[NUM_ACTIONS]** → lists with available states and actions.
- **TEMPTATION_PAYOFF**, **NUM_DEFECTS**, **P_DIFFUSOION**, **ALPHA_SHARE** → value of the temptation b, number of the defects (holes) in the lattice, probability of succesfully moving and sharing parameter s_r. All of these must be passed as arguments to the executable.
- **Q[LL][NUM_STATES][NUM_ACTIONS]** → Q table list, which will define a NUM_ACTIONS x NUM_STATES matrix for all the LL agents, i.e., LL entries in the list.
- **number_coop_average[MEASURES]**, **number_def_average[MEASURES]**, **number_coop_to_def_average[MEASURES]**, **average_Q_table[MEASURES][NUM_STATES][NUM_ACTIONS]** → lists to be used as averages through each measure.
- **s[LL]**, **payoff[LL]** → contain all the states and the payoffs of the players and will be severely accessed through the simulation.
- **right[LL]**, **left[LL]**, **top[LL]**, **down[LL]**, **neigh[LL][NUM_NEIGH]** → defines all the neighbours in each direction for every player, with the *neigh* list containing all of them.
- **empty_matrix[LL]**, **which_empty[LL]** → will account for the empty spaces in the lattice.

Functions and code funcionality

The **main** function is (clearly) first called. After doing a check that all arguments are rightly passed (with no check for not a number or such, as this is expected from you to get right), we generate a random seed and creating a time table, determining with it which steps will be measured and saved, we call the *neighbours_2d* function from *mc.h*, which gets the nearest neighbours in the left, right, ..., lists, then calling *determine_neighbours*, which fills the **neigh[LL]** list with neighbours, as such:

```
for(i=0; i<LL; ++i)
   {
       neigh[i][0] = left[i];
       neigh[i][1] = right[i];
       neigh[i][2] = top[i];
       neigh[i][3] = down[i];
   }
```

With the neighbours determined for each player, we then initialize the file, which will function as a sort of database, writing in a .dat with the name given by:

```
sprintf(output_file_freq,"data/T%.2f_S_%.2f_LSIZE%d_rho%.5f_CONF_%d_%ld_prof.dat",
                           TEMPTATION_PAYOFF,SUCKER_PAYOFF,LSIZE,1.0-NUM_DEFECTS/((float)LL),NUM_CONF,seed);
```

The condition *#ifdef USEGFX* will then check if the flag is active to display the snapshots; if not, we finally initialize the simulation, calling the main function of the code, *simulation()*. In this function, we loop through configuration indexes, which will max out at the *NUM_CONF*, thus obtaining a number of independent samples, using the for loop:

```
for (config_index = 0; config_index < NUM_CONF; ++config_index){
[...]
}
```

Inside this loop, we first initialize the environment and the players, with the *initialization()* function. After this, a for loop is responsible for dealing with the number of measures, and a while loop bootstraps the *local_dynamics(s, empty_matrix, which_empty)*. The rest of the *simulation()* function then counts the outputs from the local dynamics and saves it to the file, using again the *fprintf()* function (file printing). Let us now take a closer look at the *local_dynamics(args)* function.

Local dynamics

Inside this function, we compute a single Monte Carlo step, and thus it is where all the magic happens. After initializing all the local variables, such as the numbers of coop. and def. as well as the payoffs, we temporarily store all the initial states (in order to calculate how many cooperators turned to defectors and vice versa at the end of the mcs):

```
for (i = num_empty_sites; i < L2; ++i)
        stemp[empty_matrix[i]] = s[empty_matrix[i]];
```

Then, the main loop of starts, where for each step we start by sampling a player at random and then getting its site and state:

```
chosen_index = (int)(num_empty_sites + FRANDOM1*(LL-num_empty_sites));
        chosen_site  = empty_matrix[chosen_index];

        initial_s = s[chosen_site];
```

After, we check if the state is not 0, as the 0 state is defined as a hole. Inside this if statement, another sampling is made to decide if the player will make the decision at random or according to its Q-table:

```
if (FRANDOM1 < EPSILON) //random
     random_choice(chosen_site, &new_action, &new_action_index);
else // greedy
     find_maximum_Q_value(chosen_site, initial_s_index, &new_action, &new_action_index, &maxQ);
```

Note: important to note that we are passing the addresses of several unitialized variables, so that we can alter their values inside the functions using the pointer addresses.

The *find_maximum_Q_value(args)* function simply loops through the Q-table values and returns the action associated with the maximum value and the corresponding maximum value, e.g., maxQ.

If the action chosen in the if statement *if (new_action_index != MOVEindex)* is not moving, we update the state with *s[chosen_site] = new_action*, calculate the payoffs and obtain the reward:

```
double neighbours_payoff = get_mean_neighbours_payoff(payoff, s, chosen_site);

      final_payoff = pd_payoff(s, new_action, chosen_site);

      reward = ALPHA_SHARE * neighbours_payoff + (1 - ALPHA_SHARE) * final_payoff;
```

The *pd_payoff(args)* function simply loops through the neighbours and calculate the payoff of each interaction, summing it all, while the *get_mean_neighbours_payoff(args)* calls the *pd_payoff(args)* for each of the player's neighbours. Then, we update the Q-table according to the known formula:

```
Q[chosen_site][initial_s_index][new_action_index] +=  ALPHA * (reward + GAMMA * new_maxQ
                                      - Q[chosen_site][initial_s_index][new_action_index]);
```

In the other hand, if the decision corresponds to *MOVEindex*, we perform a random diffusion when calling the function *int rand_diffusion(args)*.

rand_diffusion

```
int rand_diffusion(int *s1, int *s, float p_diffusion,unsigned long *empty_matrix,unsigned long *which_empty)
{
    int    i, j, k, s2;
    double temp = FRANDOM1;

    if (temp < p_diffusion)
    {
    [...]
    }
```

which checks if the sampled number is less than the probability of diffusion, that is, samples if the player will succesfully move or not, then trying to move to a neighbouring site. Also, if that site is not empty, that is, the sampled site *s2 = neigh[*s1][i] != 0*, the player will not successfully move.

If it does move, all the information of the empty site s2 is exchanged with our focal player s1, including it's position. That is, we transform our original site in an empty site and copy the information of it to the previously empty one.

Finally, if the returned value from the function is 1, we have moved, so we then run this:

```
if (moved) {
      find_maximum_Q_value(chosen_site, initial_s_index, &future_action, &future_action_index, &new_maxQ);

      double neighbours_payoff = get_mean_neighbours_payoff(payoff, s, chosen_site);

      reward = ALPHA_SHARE * neighbours_payoff + (1 - ALPHA_SHARE) * final_payoff;

      Q[chosen_site][initial_s_index][new_action_index] +=  ALPHA * (reward + GAMMA * new_maxQ
                              - Q[chosen_site][initial_s_index][new_action_index] );
}
```

Which computes the same values as we did in the player did not move, but now we compute it in the new site, with new neighbours.