```
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <cassert>
#include <cmath>
#include <ctime>
#include <fstream>
#include <iostream>
#include <random>
#include <string>
#include <vector>

class Neuron
{
  int clock = 0;
  int state = 0;

 public:
  void flowing_time()
  {
    if (clock != 0) {
      clock = clock - 1;
    }
  }

  void set_clock(int time)
  {
    clock = time;
  }

  int get_clock()
  {
    return clock;
  }

  int get_state()
  {
    return state;
  }

  void set_state(int state_)
  {
    state = state_;
  }
};

double operator*(std::vector<double> l, std::vector<Neuron> r)
{
  assert(l.size() == r.size());
  double result = 0;
  auto itleft = l.begin();
  auto itright = r.begin();
  for (; itleft != l.end(); itleft++, itright++) {
    result = result + (*itleft * (*itright).get_state());
  }
  return result;
}

class Graph
```

```cpp
{
  void generate_adj()
  {
    std::random_device rd;  // Will be used to obtain a seed for the random number engine
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis_int(0, n_nodes - 1);
    double w_min = -0.5;
    double w_max = 1;
    if (EI != 0.) {
      w_max = EI * std::abs(w_min);
    }
    std::uniform_real_distribution<> dis_real(w_min, w_max);
    std::uniform_int_distribution<> dis_degree(in_degree - 2, in_degree + 1);
    assert(in_degree >= 2);

    int individual_in_degree = 0;
    auto itr = adj.begin();
    auto const endr = adj.end();
    std::vector<double> row(n_nodes);
    int N = 0;
    int r = 0;

    for (; itr != endr; ++itr, ++r) {
      individual_in_degree = dis_degree(gen);
      for (int i = 0; i != individual_in_degree; ++i) {
        N = dis_int(gen);
        if (row[N] == 0 && (adj[N])[r] == 0 && N != r) {
          row[N] = dis_real(gen);
        } else {
          --i;
        }
      }
      *itr = row;
      for (auto it = row.begin(); it != row.end(); it++) {
        *it = 0;
      }
    }
  }

 protected:
  int n_nodes = 0;
  int in_degree = 0;
  int time_active = 0;
  int time_passive = 0;
  int retard = 0;
  double EI = 0.;
  std::vector<std::vector<double>> memory;
  std::vector<std::vector<double>> adj{0};  // elemento 1 2  il link da 2 ad 1( la trasposta)
  std::vector<std::vector<double>> transpose{0};
  std::vector<Neuron> state;
  int time = 0;
  double activation_sync = 0.;

  auto get_row(int row)
  {
    assert(row < n_nodes);
    auto it_row = adj.begin();
    for (int i = 0; i != row; ++i) {
      ++it_row;
    }
```

```
    return it_row;
  }

public:
  Graph(int n_nodes_,
        int in_degree_,
        int time_active_,
        int time_passive_,
        int retard_,
        double EI_ = 0.,
        bool is_cluster = false)
  {
    n_nodes = n_nodes_;
    in_degree = in_degree_;
    time_active = time_active_ - 1;
    time_passive = time_passive_ - 1;
    retard = retard_ - 1;
    EI = EI_;
    assert(in_degree < (n_nodes - 1));
    state.resize(n_nodes);
    adj.resize(n_nodes, std::vector<double>(n_nodes));
    if (!is_cluster) {
      generate_adj();
    }
    memory.resize(retard);
    for (int i = 0; i != memory.size(); ++i) {
      memory[i].resize(n_nodes);
    }
  }

  // normalize the firing of a neuron between its "axons"
  void normalize()
  {
    int row = 0;
    int col = 0;
    double fire = 0.0;
    for (; col != n_nodes; ++col) {
      fire = 0.0;
      for (row = 0; row != n_nodes; ++row) {
        fire += std::abs(adj[row][col]);
      }
      for (row = 0; row != n_nodes; ++row) {
        if (adj[row][col] != 0) {
          adj[row][col] /= fire;
        }
      }
    }
  }

  void create_transpose()
  {
    transpose.resize(n_nodes, std::vector<double>(n_nodes));
    for (int i = 0; i < n_nodes; ++i) {
      for (int j = 0; j < n_nodes; ++j) {
        transpose[j][i] = adj[i][j];
      }
    }
  }

  // Activate each neuron with a probability of 50%. Each activated neuron starts with time_active =
```

```cpp
// 0
void random_init()
{
  std::mt19937 gen((unsigned)std::time(0));
  std::uniform_int_distribution<> dis_binary(0, 1);

  for (int i = 0; i != n_nodes; ++i) {
    int random = dis_binary(gen);
    if (random == 1) {
      state[i].set_state(1);
      state[i].set_clock(time_active);
    } else {
      state[i].set_state(0);
    }
  }
}

// Activate each neuron with probability prob. Choose random how to set the initial clock from 0
// to time_active.
void activate(double prob)
{
  assert(prob <= 1 && prob >= 0);
  std::mt19937 gen((unsigned)std::time(0));
  std::uniform_real_distribution<> prob_active(0, 1);
  std::uniform_int_distribution<> clock_active(0, time_active);
  std::uniform_int_distribution<> clock_passive(0, time_passive);

  for (int i = 0; i != n_nodes; ++i) {
    double activation = prob_active(gen);
    int time_activation = clock_active(gen);
    int time_passivation = clock_passive(gen);
    if (activation < prob) {
      state[i].set_state(1);
      state[i].set_clock(time_activation);
    } else {
      state[i].set_state(0);
      state[i].set_clock(time_passivation);
    }
  }
}

void all_firing()
{
  for (int i = 0; i != n_nodes; ++i) {
    state[i].set_state(1);
    state[i].set_clock(time_active);
  }
}

void next_step()
{
  time++;
  std::vector<double> next(n_nodes);
  auto it_next = next.begin();
  auto end = adj.end();
  for (auto it = adj.begin(); it != end; ++it) {
    *it_next = ((*it) * (state));
    it_next++;
  }
  memory.push_back(next);
```

```
  Heaviside(memory[0]);
  memory.erase(memory.begin());
}

int Heaviside(double x)
{
  if (x > 0.) {
    return 1;
  } else {
    return 0;
  }
};

void Heaviside(std::vector<double> v)
{
  auto it_state = state.begin();
  for (auto itv = v.begin(); itv != v.end(); itv++) {
    if ((*it_state).get_clock() == 0 && (*it_state).get_state() == 0) {
      (*it_state).set_state(Heaviside(*itv));
      (*it_state).set_clock(time_active);
    } else if ((*it_state).get_clock() == 0 && (*it_state).get_state() == 1) {
      (*it_state).set_state(0);
      (*it_state).set_clock(time_passive);
    }
    // Comment this else if to go back to previous dynamic
    // else if((*it_state).get_state()==0 && (*itv - (*it_state).get_clock() * 0.5) > 0){
    //    (*it_state).set_state(1);
    //    (*it_state).set_clock(time_active);
    //}
    else {
      (*it_state).flowing_time();
    }
    ++it_state;
  }
}

int get_time()
{
  return time;
}

std::vector<int> get_state()
{
  std::vector<int> int_state(state.size());
  auto its = int_state.begin();
  for (auto it = state.begin(); it != state.end(); ++it, ++its) {
    *its = (*it).get_state();
  }
  return int_state;
}

void set_state(int num_neuron, int state_)
{
  assert(num_neuron < n_nodes);
  assert(state_ == 0 || state_ == 1);
  state[num_neuron].set_state(state_);
  if (state_ == 0) {
    state[num_neuron].set_clock(time_passive);
  } else {
    state[num_neuron].set_clock(time_active);
```

```
  }
}

void set_state(std::vector<int> num_neurons, int state_)
{
  assert(state_ == 0 || state_ == 1);
  for (int i = 0; i != num_neurons.size(); ++i) {
    set_state(num_neurons[i], state_);
  }
}

void print_state()
{
  for (int i = 0; i != n_nodes; ++i) {
    std::cout << state[i].get_state() << "␣";
  }
  std::cout << '\n';
}

void print_sync()
{
  std::cout << get_sync() << '\n';
}

int out_degree(int num_neuron)
{
assert(num_neuron < n_nodes);
  int out_degree = 0;
  for(int i = 0; i < n_nodes; ++i){
    if(adj[i][num_neuron] != 0){ ++out_degree; }
  }
  return out_degree;
}

std::vector<int> num_max_out_degree_index(int num){
    std::vector<int> out_degrees(n_nodes);
    std::vector<int> out_degree_rank(n_nodes);
    int x = 0;
    for(int i = 0; i < n_nodes; ++i){
        out_degrees[i] = out_degree(i);
        out_degree_rank[i] = out_degree(i);
    }
    for(int i = 0; i < n_nodes; ++i){
        for(int j = 0; j < n_nodes - 1; ++j){
            if(out_degree_rank[j] < out_degree_rank[j + 1]){
                x = out_degree_rank[j];
                out_degree_rank[j] = out_degree_rank[j + 1];
                out_degree_rank[j + 1] = x;
            }
        }
    }
    std::vector<int> out_degree_index_rank(n_nodes);
    int cont1 = 0;
    for(int i = 0; i < n_nodes; ++i){
        if(i != 0 && out_degree_rank[i] == out_degree_rank[i - 1]){
            ++cont1;
        } else {
            cont1 = 0;
        }
        int cont2 = 0;
```

```
        for(int j = 0; j < n_nodes; ++j){
            if(out_degrees[j] == out_degree_rank[i]){
                if(cont2 == cont1){
                    out_degree_index_rank[i] = j;
                    break;
                } else{
                    ++cont2;
                }
            }
        }
    }
  out_degree_index_rank.resize(num);
    return out_degree_index_rank;
}

void instant_selective_impulse(int num_neurons){
    set_state(num_max_out_degree_index(num_neurons), 1);
}
// basic synchronization
double get_sync()
{
  double sync = 0.;
  int s;
  for (auto it = state.begin(); it != state.end(); ++it) {
    s = (*it).get_state();
    switch (s) {
      case 0:
        sync += -1;
        break;
      case 1:
        sync += 1;
        break;
      default:
        break;
    }
  }
  sync = sync / n_nodes;
  return sync;
}

double get_average_sync(int steps)
{
  for (int i = 0; i != steps; ++i) {
    next_step();
  }

  double sync_average = 0;
  for (int i = 0; i != (time_active + time_passive + 2); ++i) {
    sync_average += std::abs(get_sync());
    next_step();
  }
  return sync_average / (time_active + time_passive + 2);
}

double get_average_sync(int steps, int active, int passive, int excited)
{
  std::vector<int> excited_neurons;
  for(int i = 0; i != excited; ++i){
    excited_neurons.push_back(i);
  }
```

```
int excited_active = active;
int excited_passive = passive;
int state = 0;
int clock = excited_passive;
for(int i = 0; i != steps; ++i){
  next_step();
  set_state(excited_neurons, state);
  --clock;
  if(clock == 0){
    switch (state)
    {
    case 0:
      state = 1;
      clock = excited_active;
      break;
    case 1:
      state = 0;
      clock = excited_passive;
    break;
    default:
    std::cout << "ERROR" << '\n';
      break;
    }
  }
}

double sync_average = 0.;
for (int i = 0; i != (time_active + time_passive + 2); ++i) {
  next_step();
  set_state(excited_neurons, state);
  --clock;
  if(clock == 0){
    switch (state)
    {
    case 0:
      state = 1;
      clock = excited_active;
      break;
    case 1:
      state = 0;
      clock = excited_passive;
    break;
    default:
    std::cout << "ERROR" << '\n';
      break;
    }
  }
  sync_average += std::abs(get_sync());
}
return sync_average / (time_active + time_passive + 2);
}

double get_activation_sync(int steps)
{
  for (int i = 0; i != steps; ++i) {
    next_step();
  }

  std::vector<int> previous;
```

```
  previous.resize(n_nodes, 0);
  for (int i = 0; i < n_nodes; ++i) {
    previous[i] = state[i].get_state();
  }

  double best_activation_sync = 0.;
  double activation_sync = 0.;
  double sottract_sync = 0.;
  for (int i = 0; i < (time_active + time_passive + 2); ++i) {
    activation_sync = 0.;
    next_step();
    for (int neuron = 0; neuron < n_nodes; ++neuron) {
      if (previous[neuron] == 0) {
        activation_sync += static_cast<double>(state[neuron].get_state());
      }
      previous[neuron] = state[neuron].get_state();
    }
    if (activation_sync > best_activation_sync) {
      sottract_sync += best_activation_sync;
      best_activation_sync = activation_sync;
    } else {
      sottract_sync = sottract_sync + activation_sync;
    }
  }
  return (best_activation_sync -
          sottract_sync / static_cast<double>(time_active + time_passive + 2 - 1)) /
         static_cast<double>(n_nodes);
}

double get_activation_sync(int steps, int active, int passive, int excited)
{
  std::vector<int> excited_neurons;
  for(int i = 0; i != excited; ++i){
    excited_neurons.push_back(i);
  }

int excited_active = active;
int excited_passive = passive;
int state_ = 0;
int clock = excited_passive;
//wait steps iterations
for(int i = 0; i != steps; ++i){
  next_step();
  set_state(excited_neurons, state_);
  --clock;
  if(clock == 0){
    switch (state_)
    {
    case 0:
      state_ = 1;
      clock = excited_active;
      break;
    case 1:
      state_ = 0;
      clock = excited_passive;
    break;
    default:
    std::cout << "ERROR" << '\n';
      break;
    }
```

```
  }
}

std::vector<int> previous;
previous.resize(n_nodes, 0);
for (int i = 0; i < n_nodes; ++i) {
  previous[i] = state[i].get_state();
}

double best_activation_sync = 0.;
double activation_sync = 0.;
double sottract_sync = 0.;
for (int i = 0; i != (time_active + time_passive + 2); ++i) {
  next_step();
  set_state(excited_neurons, state_);
  --clock;
  if(clock == 0){
    switch (state_)
    {
    case 0:
      state_ = 1;
      clock = excited_active;
      break;
    case 1:
      state_ = 0;
      clock = excited_passive;
    break;
    default:
    std::cout << "ERROR" << '\n';
      break;
    }
  }

  activation_sync = 0.;
  for (int neuron = 0; neuron < n_nodes; ++neuron) {
    if (previous[neuron] == 0) {
      activation_sync += static_cast<double>(state[neuron].get_state());
    }
    previous[neuron] = state[neuron].get_state();
  }
  if (activation_sync > best_activation_sync) {
    sottract_sync += best_activation_sync;
    best_activation_sync = activation_sync;
  } else {
    sottract_sync = sottract_sync + activation_sync;
  }
}
return (best_activation_sync -
        sottract_sync / static_cast<double>(time_active + time_passive + 2 - 1)) /
       static_cast<double>(n_nodes);
}

//Return average and activation sync in a vector
std::vector<double> get_sync_vector(int steps, int active, int passive, int excited)
{
  std::vector<int> excited_neurons;
  for(int i = 0; i != excited; ++i){
    excited_neurons.push_back(i);
  }
```

```
int excited_active = active;
int excited_passive = passive;
int state_ = 0;
int clock = excited_passive;
//wait steps iterations
for(int i = 0; i != steps; ++i){
  next_step();
  set_state(excited_neurons, state_);
  --clock;
  if(clock == 0){
    switch (state_)
    {
    case 0:
      state_ = 1;
      clock = excited_active;
      break;
    case 1:
      state_ = 0;
      clock = excited_passive;
    break;
    default:
    std::cout << "ERROR" << '\n';
      break;
    }
  }
}

std::vector<int> previous;
previous.resize(n_nodes, 0);
for (int i = 0; i < n_nodes; ++i) {
  previous[i] = state[i].get_state();
}

double sync_average = 0.;
double best_activation_sync = 0.;
double activation_sync = 0.;
double sottract_sync = 0.;
for (int i = 0; i != (time_active + time_passive + 2); ++i) {
  next_step();
  set_state(excited_neurons, state_);
  --clock;
  if(clock == 0){
    switch (state_)
    {
    case 0:
      state_ = 1;
      clock = excited_active;
      break;
    case 1:
      state_ = 0;
      clock = excited_passive;
    break;
    default:
    std::cout << "ERROR" << '\n';
      break;
    }
  }

  activation_sync = 0.;
  for (int neuron = 0; neuron < n_nodes; ++neuron) {
```

```
      if (previous[neuron] == 0) {
        activation_sync += static_cast<double>(state[neuron].get_state());
      }
      previous[neuron] = state[neuron].get_state();
    }
    if (activation_sync > best_activation_sync) {
      sottract_sync += best_activation_sync;
      best_activation_sync = activation_sync;
    } else {
      sottract_sync = sottract_sync + activation_sync;
    }
    sync_average += std::abs(get_sync());
  }

  //first place-> average_sync
  //second place-> activation_sync
  std::vector<double> return_sync;
  return_sync.push_back(sync_average / (time_active + time_passive + 2));
  return_sync.push_back((best_activation_sync -
          sottract_sync / static_cast<double>(time_active + time_passive + 2 - 1)) /
         static_cast<double>(n_nodes));

  return return_sync;
  }

  //Return average and activation syncs in a vector, with an instant impulse
  std::vector<double> get_sync_vector(int steps, int impulse){
      instant_selective_impulse(impulse);

      for (int i = 0; i != steps; ++i) {
        next_step();
      }

      double sync_average = 0;
      for (int i = 0; i != (time_active + time_passive + 2); ++i) {
        sync_average += std::abs(get_sync());
        next_step();
      }

      std::vector<int> previous;
      previous.resize(n_nodes, 0);
      for (int i = 0; i < n_nodes; ++i) {
        previous[i] = state[i].get_state();
      }

      double best_activation_sync = 0.;
      double activation_sync = 0.;
      double sottract_sync = 0.;
      for (int i = 0; i < (time_active + time_passive + 2); ++i) {
        activation_sync = 0.;
        next_step();
        for (int neuron = 0; neuron < n_nodes; ++neuron) {
          if (previous[neuron] == 0) {
            activation_sync += static_cast<double>(state[neuron].get_state());
          }
          previous[neuron] = state[neuron].get_state();
        }
        if (activation_sync > best_activation_sync) {
          sottract_sync += best_activation_sync;
          best_activation_sync = activation_sync;
```

```
      } else {
        sottract_sync = sottract_sync + activation_sync;
      }
    }


    //first place-> average_sync
    //second place-> activation_sync
    std::vector<double> return_sync;
    return_sync.push_back(sync_average / (time_active + time_passive + 2));
    return_sync.push_back((best_activation_sync -
        sottract_sync / static_cast<double>(time_active + time_passive + 2 - 1)) /
       static_cast<double>(n_nodes));
    return return_sync;
}

// CSV file. First column = vertices; second column = step number; third column = evolution of
// first vertex, ... To create graph, upload file on mathcha(Complex_systems)
void write_CSV(int num_visible, int num_interactions)
{
  std::mt19937 gen((unsigned)std::time(0));
  std::uniform_int_distribution<> choose_neurons(0, n_nodes - 1);
  std::vector<int> visible;
  int neuron_visible;
  assert(num_visible <= n_nodes);

  // extract nodes and memorize them in a sorted vector
  for (int i = 0; i != num_visible; ++i) {
    neuron_visible = choose_neurons(gen);
    auto it = std::find(visible.begin(), visible.end(), neuron_visible);
    if (it == visible.end()) {
      visible.push_back(neuron_visible);
    } else {
      --i;
    }
  }
  std::sort(visible.begin(), visible.end());

  // Write CSV file
  int stop = num_interactions;
  if (num_visible > num_interactions) {
    stop = num_visible;
  }
  std::ofstream SaveFile("CSV.txt");
  for (int i = 0; i != stop; ++i) {
    if (i < num_visible) {
      SaveFile << "V" << visible[i] << ",";
    } else {
      SaveFile << ",";
    }
    if (i < num_interactions) {
      SaveFile << i << ",";
      for (int vertex = 0; vertex != num_visible; ++vertex) {
        SaveFile << state[visible[vertex]].get_state() << ",";
      }
      SaveFile << '\n';
    } else {
      SaveFile << '\n';
    }
      next_step();
```

```
    }
    SaveFile.close();
  }

  void write_adj_txt()
  {
    auto itr = adj.begin();
    auto const endr = adj.end();
    auto endc = itr->end();
    auto itc = itr->begin();
    std::ofstream SaveFile("Matrix.txt");
    SaveFile << "Adjacency␣Matrix" << '\n';
    for (; itr != endr; ++itr) {
      itc = itr->begin();
      endc = itr->end();
      for (; itc != endc; ++itc) {
        SaveFile << *itc << '\n';  // this function prints the adj matrix in a txt file
      }
    }
    SaveFile << '\n';
    SaveFile.close();
  }

  void read_adj_txt()
    {
    auto itr = adj.begin();
    auto const endr = adj.end();
    auto endc = itr->end();
    auto itc = itr->begin();
    std::string val;

    std::ifstream inFile;
    inFile.open("Matrix.txt");
    while (getline(inFile, val)) {
      for (; itr != endr; ++itr) {
        itc = itr->begin();
        endc = itr->end();
        for (; itc != endc; ++itc) {
          inFile >> val;
          (*itc) = stod(val);
        }
      }
    }
    inFile.close();
  }
};

class Bipartite : public Graph
{
  void bipartite_adj()
  {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis_int(0, n_nodes / 2 - 1);
    std::uniform_real_distribution<> dis_real(10, 20);

    // Create a bipartite adjacency matrix
    auto itr = adj.begin();
    auto const endr = adj.end();
    auto itc = itr->begin();
```

```
    auto endc = itr->end();
    std::vector<double> row(n_nodes);
    int N = 0;
    int r = 0;
    int half = 1;
    for (; itr != endr; ++itr, ++r) {
      if (r >= n_nodes / 2) {
        half = 0;
      }
      for (int i = 0; i != in_degree; ++i) {
        N = dis_int(gen);
        if (row[N + half * n_nodes / 2] == 0 && (adj[N + half * n_nodes / 2])[r] == 0) {
          row[N + half * n_nodes / 2] = dis_real(gen);
        } else {
          --i;
        }
      }
      *itr = row;
      for (auto it = row.begin(); it != row.end(); it++) {
        *it = 0;
      }
    }
  }

public:
  Bipartite(int n_nodes_, int in_degree_, int time_active_, int time_passive_, int retard_)
      : Graph(n_nodes_, in_degree_, time_active_, time_passive_, retard_, 0., true)
  {
    bipartite_adj();
  }

  double get_sync()
  {
    double sync_2 = 0;
    double sync_1 = 0;
    int s;
    for (int i = 0; i != n_nodes; ++i) {
      s = state[i].get_state();
      if (i == n_nodes / 2) {
        sync_1 = sync_2;
        sync_2 = 0;
      }
      switch (s) {
        case 0:
          sync_2 += -1;
          break;
        case 1:
          sync_2 += 1;
          break;
        default:
          break;
      }
    }
    return (std::abs(sync_1) + std::abs(sync_2)) / n_nodes;
  }

  void print_syncs()
  {
    double sync_2 = 0;
    double sync_1 = 0;
```

```
    int s;
    for (int i = 0; i != n_nodes; ++i) {
      s = state[i].get_state();
      if (i == n_nodes / 2) {
        sync_1 = sync_2;
        sync_2 = 0;
      }
      switch (s) {
        case 0:
          sync_2 += -1;
          break;
        case 1:
          sync_2 += 1;
          break;
        default:
          break;
      }
    }
    std::cout << sync_1 / n_nodes * 2 << "␣␣" << sync_2 / n_nodes * 2 << '\n';
  }

  void print_sync()
  {
    std::cout << get_sync() << '\n';
  }

  void bias_init(double p_1, double p_2)
  {
    std::mt19937 gen((unsigned)std::time(0));
    std::uniform_real_distribution<> dis(0.0, 1.0);

    double p = p_1;
    for (int i = 0; i != n_nodes; ++i) {
      double random = dis(gen);
      if (random < p) {
        state[i].set_state(1);
        state[i].set_clock(time_active);
      }
      if (i == n_nodes / 2) {
        p = p_2;
      }
    }
  }
};

class Cluster : public Graph
{
  int num_clusters = 0;
  int nodes_in_cluster = 0;
  int inter_connections = 0;

  void cluster_adj()
  {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis_int(0, nodes_in_cluster - 1);
    std::uniform_real_distribution<> dis_real(5, 10);            // intracluster
    std::uniform_real_distribution<> dis_inhibitor(-150, -140);  // intercluster

    // Create connections inside the clusters
```

```cpp
    auto itr = adj.begin();
    auto endr = adj.end();
    auto itc = itr->begin();
    auto endc = itr->end();
    std::vector<double> row(n_nodes);
    int N = 0;
    int r = 0;
    int j = 0;
    for (int index_cluster = 0; index_cluster != num_clusters; ++index_cluster) {
      for (int j = 0; j != nodes_in_cluster; ++j, ++itr, ++r) {
        for (int i = 0; i != in_degree; ++i) {
          N = dis_int(gen);
          if (row[N + index_cluster * nodes_in_cluster] == 0 &&
              (adj[N + index_cluster * nodes_in_cluster])[r] == 0 &&
              (N + index_cluster * nodes_in_cluster) != r) {
            row[N + index_cluster * nodes_in_cluster] = dis_real(gen);
          } else {
            --i;
          }
        }
        *itr = row;
        for (auto it = row.begin(); it != row.end(); it++) {
          *it = 0;
        }
      }
      // Create inhibitory connections between the clusters
      for (int end_cluster = 0; end_cluster != num_clusters; ++end_cluster) {
        if (end_cluster == index_cluster) {
          // Do nothing, skip.
        } else {
          for (int i = 0; i != inter_connections; ++i) {
            N = dis_int(gen);
            int C = dis_int(gen);
            if ((adj[C + index_cluster * nodes_in_cluster])[N + end_cluster * nodes_in_cluster] ==
                    0 &&
                (adj[N + end_cluster * nodes_in_cluster])[C + index_cluster * nodes_in_cluster] ==
                    0) {
              (adj[C + index_cluster * nodes_in_cluster])[N + end_cluster * nodes_in_cluster] =
                  dis_inhibitor(gen);
            } else {
              --i;
            }
          }
        }
      }
    }
  }

public:
  Cluster(int num_clusters_,
          int nodes_in_cluster_,
          int inter_connections_,
          int in_degree_,
          int time_active_,
          int time_passive_,
          int retard_)
      : Graph(num_clusters_ * nodes_in_cluster_,
              in_degree_,
              time_active_,
              time_passive_,
```

```
            retard_,
            0.,
            true)
{
  inter_connections = inter_connections_;
  num_clusters = num_clusters_;
  nodes_in_cluster = nodes_in_cluster_;
  cluster_adj();
}

  void activate(double prob){
    assert(prob <= 1 && prob >= 0);
    std::mt19937 gen((unsigned) std::time(0));
    std::uniform_real_distribution<> prob_active(0, 1);
    std::uniform_int_distribution<> clock_active(0, time_active);
    std::uniform_int_distribution<> clock_passive(0, time_passive);

    for(int i = 0; i != nodes_in_cluster; ++i){
      double activation = prob_active(gen);
      int time_activation = clock_active(gen);
      int time_passivation = clock_passive(gen);
      if( activation < prob){
        state[i].set_state(1);
        state[i].set_clock(time_activation);
      } else {
        state[i].set_state(0);
        state[i].set_clock(time_passivation);
      }
    }
  }

void print_sync()
{
  double sync = 0;
  int s;
  int i = 1;
  for (auto it = state.begin(); it != state.end(); ++it, ++i) {
    s = (*it).get_state();
    switch (s) {
      case 0:
        sync += -1;
        break;
      case 1:
        sync += 1;
        break;
      default:
        break;
    }
    if (i == nodes_in_cluster) {
      i = 0;
      std::cout << sync / nodes_in_cluster << "   ";
      sync = 0;
    }
  }
  std::cout << '\n';
}

void cycle()
{
  for (int row = 0; row != n_nodes; ++row) {
```

```
      for (int col = 0; col != n_nodes; ++col) {
        if (adj[row][col] != 0) {
          adj[row][col] = -adj[row][col];

          if (!((((int)col / nodes_in_cluster) == ((int)row / nodes_in_cluster) + 1 ||
                 ((int)col / nodes_in_cluster) == 0 &&
                     ((int)row / nodes_in_cluster) == (num_clusters - 1)) ||
                ((int)col / nodes_in_cluster + 1) == ((int)row / nodes_in_cluster + 1))) {
            adj[row][col] = 0;
          }
        }
      }
    }
  }
};

int main()
{
  int time_active = 2;
  int time_passive = 1;
  int retard = 3;
  int number_neurons = 105;
  int in_degree = 6;
  double EI = 2.5;
  int num_cluster = 3;
  int nodes_in_cluster = 50;
  int intercluster = 200;
  /**/

  // Graph G(number_neurons, in_degree, time_active, time_passive, retard, EI);
   //Cluster G(num_cluster, nodes_in_cluster, intercluster, in_degree, time_active, time_passive,
    retard);
  //Bipartite G(number_neurons, in_degree, time_active, time_passive, retard);
  //
  // G.normalize();
  // G.activate(0.66);
  // G.write_adjlist();
  // G.cycle();
  // G.print_adj();
  // G.all_firing();
  // G.bias_init(0.2, 0.);
  // G.random_init();
  // G.print_adj();
  // G.create_transpose();
  // G.print_transpose();
  // G.print_adj_txt();
  // G.write_adj();
  //G.write_CSV(40, 400);
  // G.average_sync(1);

  // G.average_sync(50);
  // std::cout << "Activation_sync: " << G.get_activation_sync(50) << '\n' << '\n';
  // for (int i = 0; i != 700; ++i) {
  //    G.print_state();
  //    G.print_sync();
  //    G.next_step();
  //}

  // Per grafici cambiando i parametri: grafo classico
  /**/
```

```
double par_min = 0.;
double par_max = 1.0000001;
double par_jump = 0.1;
int steps = 200;
int N = 5;
double meta_average_sync = 0.;
double meta_activation_sync = 0.;
std::vector<double> average_sync;
std::vector<double> activation_sync;
double E_average_sync = 0.;
double E_activation_sync = 0.;

std::ofstream SaveFile_1("average_sync.txt");
std::ofstream SaveFile_2("activation_sync.txt");
std::ofstream SaveFile_3("ROOT.txt");
//std::ofstream SaveFile_4("E_average_sync.txt");
std::ofstream SaveFile_5("E_activation_sync.txt");
SaveFile_1 << "par, sync" << '\n';
SaveFile_2 << "par, sync" << '\n';
SaveFile_5 << "par, sync, error" << '\n';
for(double par = par_min; par <= par_max; par += par_jump){
  Graph G(number_neurons, in_degree, time_active, time_passive, retard, EI);
  G.normalize();
  meta_average_sync = 0.;
  meta_activation_sync = 0.;
  average_sync.resize(0);
  activation_sync.resize(0);
  E_average_sync = 0.;
  E_activation_sync = 0.;
  std::vector<double> sync_vector(2);
  for(int i = 0; i != N; ++i){
    G.activate(0.3);
    for(int wait = 0; wait != 50; ++wait){
      G.next_step();
    }
    for(int impulse = 0; impulse != 10; ++impulse){
      G.instant_selective_impulse(par * number_neurons);
      for(int wait = 0; wait != 21; ++wait){
        G.next_step();
      }
    }
    //sync_vector = G.get_sync_vector(steps, par * number_neurons);
    sync_vector[0] = G.get_average_sync(0);
    sync_vector[1] = G.get_activation_sync(0);
    average_sync.push_back(sync_vector[0]);
    meta_average_sync += average_sync[i];
    activation_sync.push_back(sync_vector[1]);
    meta_activation_sync += activation_sync[i];
  }
  SaveFile_1 << par << ", " << meta_average_sync/N << '\n';
  SaveFile_2 << par << ", " << meta_activation_sync/N << '\n';
  for(int i = 0; i != N; ++i){
    E_average_sync += (meta_average_sync / N - average_sync[i]) * (meta_average_sync / N -
average_sync[i]);
 E_activation_sync += (meta_activation_sync / N - activation_sync[i]) *
(meta_activation_sync / N - activation_sync[i]);
  }
  SaveFile_3 << par << '\t' << meta_average_sync/N << '\t' << std::sqrt(E_average_sync / (N - 1))
<< '\n'; SaveFile_5 << par << ", " << meta_activation_sync/N << ", " <<
std::sqrt(E_activation_sync / (N - 1)) << '\n';
```

```
  }
  SaveFile_1.close();
  SaveFile_2.close();
  SaveFile_3.close();
  SaveFile_5.close();
  /**/
}
```