```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>
#include <fstream>
#include <sys/time.h>
#include <cstdlib>
#include <float.h>
#include <random>
#include <stdint.h>
#include <climits>
#include <vector>
#include <string.h>

#include <omp.h>
#include <chrono>
#include <ctime>

#define _USE_MATH_DEFINES
using namespace std;

#define NUM_SAMPLES 20

/*
 SETUP:
 - hitI pick uniformly from 2ms - 8ms (Y)
 - X: 400 time series

 Fix everything else.
 */



int NE = 300;
int NI = 100;
int Level = 100;
double PEE = 0.15;
double PIE = 0.5;
double PEI = 0.5;
double PII = 0.4;

double SEE = 5.0;
double SIE = 2.0;
double SEI = -3.0;
double SII = -3.0;


double Ref = 250.0;
```

```
double HitE = 200.0;

int Reverse = -66;
double kickE = 2000.0; // lambda_E
double kickI = 2000.0; // lambda_I
double terminate_time = 1.0;


int real2int(const double x, mt19937& mt, uniform_real_distribution<double>& u)
{
    int xf = floor(x);
    double q = x - (double)xf;
    double y = 0;
    if( u(mt) < q)
        y = xf + 1;
    else
        y = xf;
    return y;
}

template <class T>
class Vector : public vector<T> {
private:
    T sum;
public:

    double get_sum()
    {
        return sum;
    }

    void maintain()
    {
        T tmp_sum = 0;
        for(auto && it : (*this))
            tmp_sum += it;
 //       cout<<tmp_sum<<endl;
        sum = tmp_sum;
    }

    void switch_element(const int index, T element)
    //switch [index] by element and maintain sum
    {
        sum = sum - (*this)[index] + element;
        (*this)[index] = element;
    }
    //The first three function is only for Clock[]

    bool remove_elt(unsigned index)
```

```
//remove element [index]
{
    if(this -> empty() || index < 0 || index >= this -> size()) {
        cout<<"Empty vector"<<endl;
        return false;
    }
    if(index == this->size() - 1) {
        this->pop_back();
    }
    else {
        T elt = (*this)[(*this).size() - 1];
        (*this)[index] = elt;
        (*this).pop_back();
    }
    return true;
}

int select(mt19937& mt, uniform_real_distribution<double>& u)
{
    if( this->size() == 0)
    {
        cout<<"Size cannot be zero "<<endl;
        return -1;
    }
    else
    {
        int index = floor(u(mt)*this->size());
        T tmp =  (*this)[index];
        remove_elt(index);
        return tmp;
    }
}

int find_min() {
    if(!this -> empty()) {
        auto min = (*this)[0];
        auto index = 0;
        auto iter = 0;
        for(auto&& item : (*this)) {
            if(item < min) {
                min = item;
                index = iter;
            }
            iter++;
        }
        return index;
    }
    return -1;
}

void print_vector() {
```

```
        for(auto && it : (*this)) {
            cout<<it<<" ";
        }
        cout<<endl;
    }

};


int find_index(Vector<double>& array, mt19937& mt,
 uniform_real_distribution<double>& u)
{
    double sum = array.get_sum();
    double tmp = u(mt);
    int index = 0;
    int size = array.size();
    double tmp_double = array[0]/sum;
//    cout<<tmp_double<<" "<<tmp<<endl;
    while(tmp_double < tmp && index < size - 1)
    {
        index++;
        tmp_double+=array[index]/sum;
    }
    return index;
}

void spikeE(const int whichHit, Vector<double>& Clock, vector<int> &VE,
 Vector<int> &HEE, Vector<int> &HIE, Vector<int> &Eref, vector<int> &awakeE,
 vector<int> &awakeI, mt19937& mt, uniform_real_distribution<double>& u)
{
    VE[whichHit] = 0;
    awakeE[whichHit] = 0;
    Eref.push_back(whichHit);
    Clock.switch_element(6, Ref*Eref.size());
    for(int i = 0; i < NE; i++)
    {
        // independently flipping coin for every neuron.
        if(u(mt) < PEE && awakeE[i])
        {
            HEE.push_back(i);
        }
    }
    for(int i = 0; i < NI; i++)
    {
        if(u(mt) < PIE && awakeI[i])
        {
            HIE.push_back(i);
        }
    }
    Clock.switch_element(2, HitE*HEE.size());
    Clock.switch_element(3, HitE*HIE.size());
```

```
}



void spikeI(const int whichHit, Vector<double>& Clock, vector<int> &VI,
 Vector<int> &HEI, Vector<int> &HII, Vector<int> &Iref, vector<int> &awakeE,
 vector<int> &awakeI, mt19937& mt, uniform_real_distribution<double>& u,
 double HitI)
{
    VI[whichHit] = 0;
    awakeI[whichHit] = 0;
    Iref.push_back(whichHit);
    Clock.switch_element(7, Ref*Iref.size());
    for(int i = 0; i < NE; i++)
    {
        if(u(mt) < PEI && awakeE[i])
        {
            HEI.push_back(i);
        }
    }
    for(int i = 0; i < NI; i++)
    {
        if(u(mt) < PII && awakeI[i])
        {
            HII.push_back(i);
        }
    }
    Clock.switch_element(4, HitI*HEI.size());
    Clock.switch_element(5, HitI*HII.size());
}

void update(vector<int> &result, vector<double> &time_spike, vector<int>
 &num_spike, double HitI, mt19937& mt, uniform_real_distribution<double>& u,
 Vector<double>& Clock,
        vector<int> &VE, vector<int> &VI, Vector<int> &HEE, Vector<int> &HEI,
         Vector<int> &HIE, Vector<int> &HII,
        Vector<int> &Eref, Vector<int> &Iref, vector<int> &awakeE, vector<int>
         &awakeI)
{
    double current_time = 0.0;
    int count = 0;
    while(current_time < terminate_time)
    {
        current_time += -log(1 - u(mt))/Clock.get_sum();
        int index =         find_index(Clock, mt, u);
        int whichHit;
        count ++;
        int local_index;
```

```
int  bin_num = (int)floor(current_time*200) % 200;
switch (index)
{
    case 0:
        whichHit = floor(u(mt)*NE);
        if(awakeE[whichHit])
        {
            VE[whichHit]++;
            if(VE[whichHit] >= Level)
            {
                spikeE(whichHit, Clock, VE, HEE, HIE, Eref, awakeE,
                 awakeI, mt, u);
                 result[bin_num] ++;

                time_spike.push_back(current_time);
                num_spike.push_back(whichHit);

            }
        }
        break;
    case 1:
        whichHit = floor(u(mt)*NI);
        if(awakeI[whichHit])
        {
            VI[whichHit]++;
            if(VI[whichHit]>= Level)
            {
                bin_num = bin_num + 200;
                spikeI(whichHit, Clock, VI, HEI, HII, Iref, awakeE,
                 awakeI, mt, u, HitI);

                 result[bin_num] ++;

                time_spike.push_back(current_time);
                num_spike.push_back(whichHit + NE);

            }
        }
        break;
    case 2:
        whichHit = HEE.select(mt, u);
        if(awakeE[whichHit])
        {

            VE[whichHit] += real2int(SEE, mt, u);
            if( VE[whichHit] >= Level)
            {
                spikeE(whichHit, Clock, VE, HEE, HIE, Eref, awakeE,
                 awakeI, mt, u);
                 result[bin_num] ++;
```

```
                    time_spike.push_back(current_time);
                    num_spike.push_back(whichHit);
                }
            }
            Clock.switch_element(2, HitE*HEE.size());
            break;
        case 3:
            whichHit = HIE.select(mt, u);
//              cout<<HEE.size()<<endl;
            if(awakeI[whichHit])
            {
                VI[whichHit] += real2int(SIE, mt, u);
                if( VI[whichHit] >= Level)
                {
                    bin_num = bin_num + 200;
                    spikeI(whichHit, Clock, VI, HEI, HII, Iref, awakeE,
                     awakeI, mt, u, HitI);
                     result[bin_num] ++;

                    time_spike.push_back(current_time);
                    num_spike.push_back(whichHit + NE);
                }
            }
            Clock.switch_element(3, HitE*HIE.size());
            break;
        case 4:
            whichHit = HEI.select(mt, u);
            if(awakeE[whichHit])
            {
                VE[whichHit] += real2int(SEI, mt, u);
                if(VE[whichHit] < Reverse)
                    VE[whichHit] = Reverse;
            }
            Clock.switch_element(4, HitI*HEI.size());
            break;
        case 5:
            whichHit = HII.select(mt, u);
            if(awakeI[whichHit])
            {
                VI[whichHit] += real2int(SII, mt, u);
                if(VE[whichHit] < Reverse)
                    VE[whichHit] = Reverse;
            }
            Clock.switch_element(5, HitI*HII.size());
            break;
        case 6:
            whichHit = Eref.select(mt, u);
            awakeE[whichHit] = 1;
            Clock.switch_element(6, Ref*Eref.size());
            break;
        case 7:
```

```
                whichHit = Iref.select(mt, u);
                awakeI[whichHit] = 1;
                Clock.switch_element(7, Ref*Iref.size());
                break;
        }
    }
}


void simulate(int thread_num, vector<int> &result, vector<double> &time_spike,
 vector<int> &num_spike, double hitI){

    time_t seconds = time(NULL);
    mt19937 mt(seconds + (double)thread_num);

    uniform_real_distribution<double> u(0, 1);

    Vector<int> VE, VI;
    VE.reserve(NE);
    VI.reserve(NI);


    for(auto i : VE)
        i = 0;
    for(auto i : VI)
        i = 0;

    Vector<int> HEE, HEI, HII, HIE, Eref, Iref;
    HEE.reserve(100000);
    HEI.reserve(100000);
    HII.reserve(100000);
    HIE.reserve(100000);
    Eref.reserve(NE);
    Iref.reserve(NI);
    vector<int> awakeE(NE);
    for(auto & i : awakeE)
        i = 1;
    vector<int> awakeI(NI);
    for(auto & i : awakeI)
        i = 1;
    Vector<double> Clock;
    Clock.reserve(8);
    //0 Edrive, 1 Idrive, 2 HEE, 3, HEI, 4, HIE, 5, HII, 6, Eref, 7, Iref
    Clock.push_back(NE*kickE);
    Clock.push_back(NI*kickI);
    for(int i = 2; i < 8; i++)
        Clock.push_back(0);
    Clock.maintain();

    update(result, time_spike, num_spike, hitI, mt, u,  Clock,
               VE, VI, HEE, HEI, HIE, HII,
```

```
                Eref, Iref, awakeE, awakeI);
}


void get_random(double* randNumbers, double low, double high, int N){
    int numThreads;
    uniform_real_distribution<double> u(0, 1);


    #pragma omp parallel
    {
        time_t seconds = time(NULL);
        int thread_num = omp_get_thread_num();
        numThreads = omp_get_num_threads();
        mt19937 mt(seconds + (double)thread_num);
            #pragma omp for
            for(int i=0; i < N; i++){
             randNumbers[i] = low+(high-low)*u(mt);
            }
    }
    }


    // NEED TO CHANGE
    // format X, y \n
    // hitIs, y \n
void write_to_disk(vector<vector<int> > &results, vector<vector<int> >
 &num_spike,
                   vector<vector<double> > &time_spike, double* hitIs, int N){
        char* file_name = "./dataHitI_smallDrive/varyHitI_2000.txt";
        ofstream myfile;
        myfile.open(file_name);

        for (int i=0; i < N; i++){
            myfile << hitIs[i] << " ";
            // cout << "hitI" << hitIs[i] << endl;
            for (double x: results[i]){
                // cout << "x: " << x << endl;
                myfile << x << " ";
            }
            myfile << endl;
        }

    myfile.close();

    char generic_file[] = "varyHitI_init_.txt";
    string base(".txt");

        for (int k=0; k < N; k++){
//            ofstream curfile;
```

```
//            cout << "varyHitI" + to_string(k) + base << endl;

//            temp_str = strcat(strcat(generic_file, temp_str), ".txt");

            myfile.open("./dataHitI_smallDrive/varyHitI" + to_string(k) + base
             );

            for (int j = 0; j < time_spike[k].size(); j ++){
//                cout << time_spike[k] << end << endl;
                myfile<<time_spike[k][j]<<"  "<<num_spike[k][j]<<endl;
            }

            myfile.close();

        }

    }


    void generateData(vector<vector<int> > &results, vector<vector<int> >
     &num_spike,
                      vector<vector<double> > &time_spike, double hitIs[]){



        #pragma omp parallel for
        for(int i=0; i < NUM_SAMPLES; i++){
            int thread_num = omp_get_thread_num();
            double hitI = hitIs[i];

            vector<int> res_arr(400, 0);
            vector<int> num_spike_arr;
            vector<double> time_spike_arr;
            num_spike_arr.reserve(10000);
            time_spike_arr.reserve(10000);




            simulate(thread_num, res_arr, time_spike_arr, num_spike_arr, hitI);

            results[i] = res_arr;
            num_spike[i] = num_spike_arr;
            time_spike[i] = time_spike_arr;

            // results[i] = vector<double>(4, 0.2);
        }
    }
```

```
    int main()
    {

        float low = 200;
        float high = 800;

        struct timeval t1, t2;
        gettimeofday(&t1,NULL);


        double hitIs[NUM_SAMPLES];
        get_random(hitIs, low, high, NUM_SAMPLES);


        vector<vector<int> > results(NUM_SAMPLES);
        vector<vector<int> > num_spike(NUM_SAMPLES);
        vector<vector<double> > time_spike(NUM_SAMPLES);


        generateData(results, num_spike, time_spike, hitIs );


    gettimeofday(&t2, NULL);
    double delta = ((t2.tv_sec  - t1.tv_sec) * 1000000u +
                    t2.tv_usec - t1.tv_usec) / 1.e6;

    write_to_disk(results, num_spike, time_spike, hitIs, NUM_SAMPLES);

    cout << "total CPU time = " << delta <<endl;


}
```