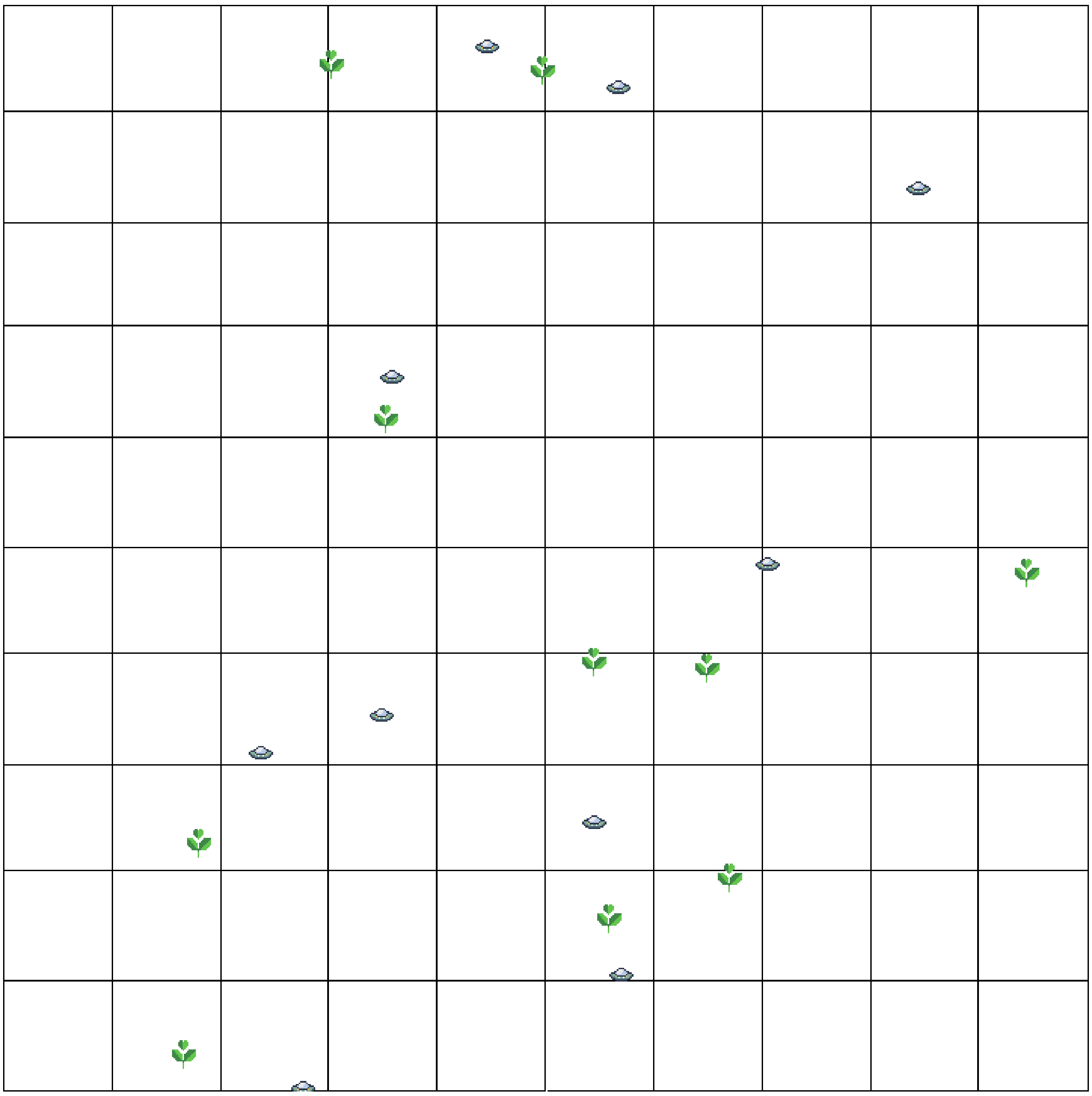
Search for weed in crops using multi-agent systems: Analytic modelling

Adrien Komaroff-Kourloff[1]

[1]Ecole des Mines de Saint-Etienne ISMIN / University of Oslo

November 2022



Contents

List of Figures

Introduction

This document is an assignment report based on the TEK5010 course at UiO. Reader is assumed to have read the assignment documents. It builds on the previous assignment documents
In that document, a simulation of a multi-agent system aimed at collecting weed-crops is being performed. The agents are very simple (move randomly without any navigation system). This time a wider search area of 10 km^2 is used, and the weed-crops detection radius is now $Tr = 0.5$ m. The speed of the agent remain unchanged : $V = 10$ km/h
One can find the source code on github.

1 N = 1 Agent, changes in simulation (Q1)

Figure 1 presents how the simulation step affected the effective search area in previous assignment. Indeed, only the points within radius range Tr were checked after the movement had been done.

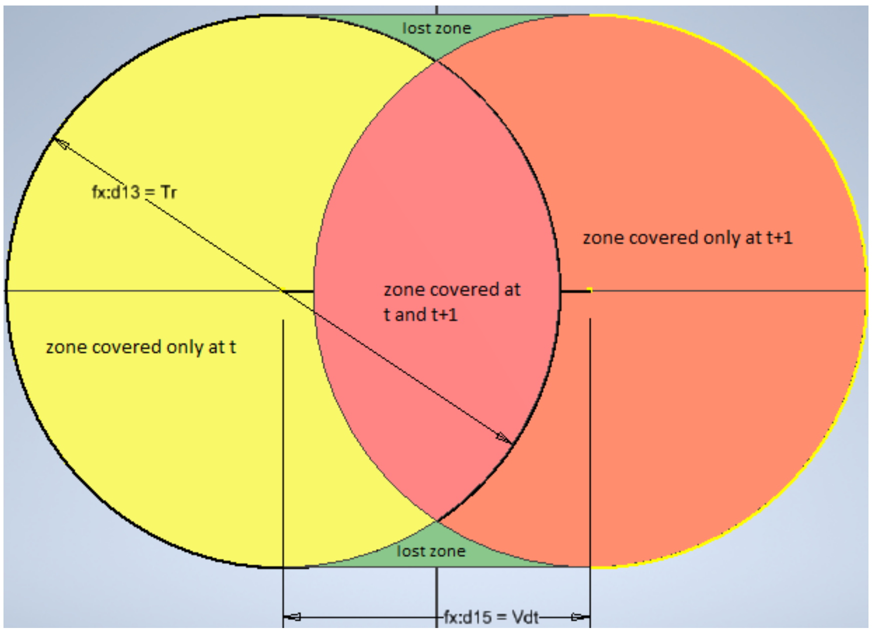


Figure 1: Effect of simulation step

Figure 2 shows how for the same spatial period the green lost zone become more prominent.

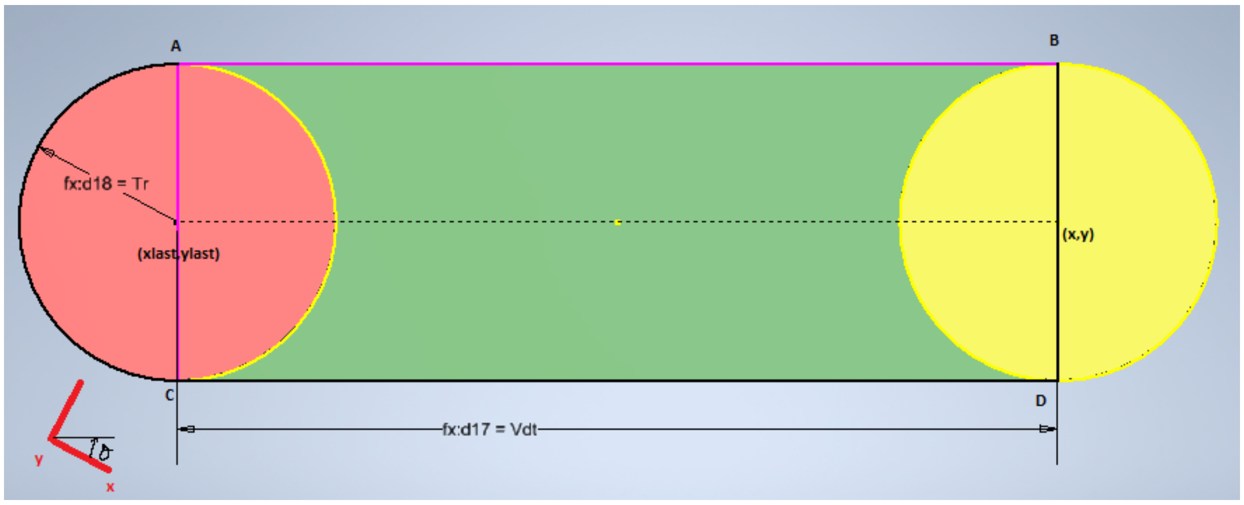


Figure 2: Effect of simulation step with smaller Radius

At this point it is possible to reduce the simulation step to get finer spatial period. It will nonetheless change the movement dynamics compared to Assignment 1 and will be heavier in terms of computation.

That is why the simulation now additionally check the presence of a weed-crop in the green zone : In fact it is the presence of a weed-crop in the rectangle ABCD which is checked In order to do so, each agent store its previous position and the weed-crop position is being checked with :

$$A = (lastx * cos(theta) + lasty * sin(theta), lasty * cos(theta) - lastx * sin(theta) + Tr)$$

$$B = (x * cos(theta) + y * sin(theta), y * cos(theta) - x * sin(theta) + Tr)$$

$$C = (lastx * cos(theta) + lasty * sin(theta), lasty * cos(theta) - lastx * sin(theta) - Tr)$$

Then for each crop :

$$xCropframeABCD = lasty * cos(theta) - lastx * sin(theta)$$

$$yCropframeABCD = lasty * cos(theta) - lastx * sin(theta)$$

Finally the following check enable to detect weed-crop in ABCD :

$$xCropframeABCD \in [A_x, B_x]$$

$$yCropframeABCD \in [C_y, A_y]$$

With those changes a simulation of 1 Agent in the field has been conducted. The simulation has been stopped after 100 task were completed :

N	Mean	std	Pexp	LB95	UB 95	Ptheory
1	6,11E+06	4,64E+06	0	5,18E+06	7,04E+06	9,41E-02

Corresponding to :

- N : number of Agent.
- Mean : mean in seconds for a crop to be found.
- std : standard deviation.
- Pexp : experimental estimated probability of a task being collected in less than 1 hour.
- LB95 Assuming Gaussian distribution of the finding time, the lower bound of confidence interval.
- UB95 Assuming Gaussian distribution of the finding time, the upper bound of confidence interval.
- Ptheory : theoretical probability of a task being collected in less than 1 hour.

2 Completing task in less than 1 hour (Q2)

Thanks to the simulation changed introduced in (Q1) it has been possible to compute the result directly.

Figure 3 shows the average time to complete the task with the number of agent varying. The average were computed with 100 samples.

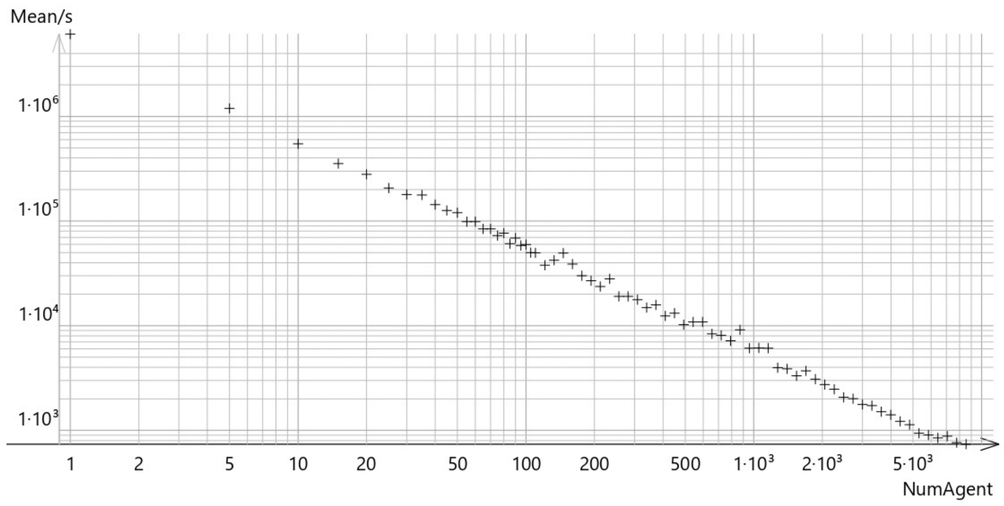


Figure 3: Mean completion time with number of agents varying

Figure 4 shows an affine regression relative to the log values in Figure 3.

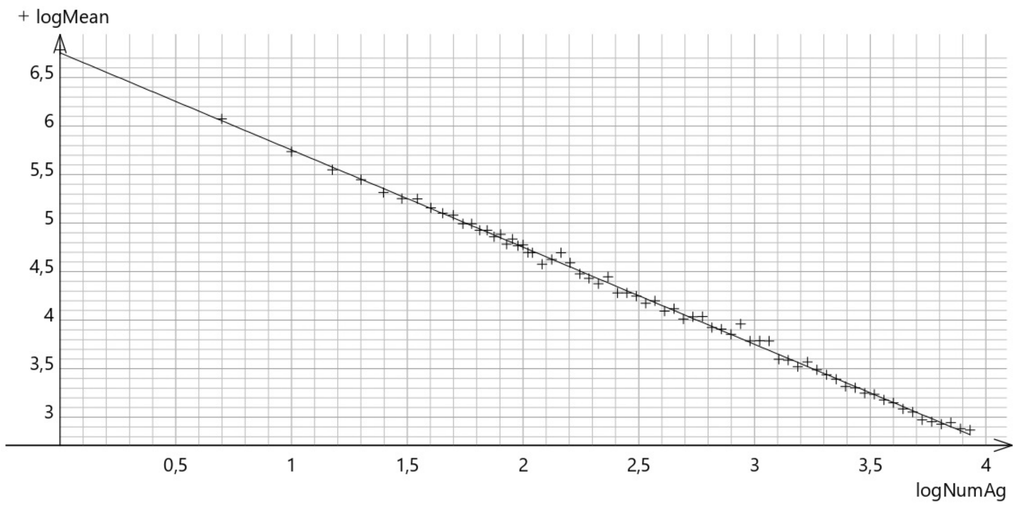


Figure 4: Mean completion time with number of agents varying

With :

$$log(Mean) = a * log(N) + b$$
$$a = (-1,001 \pm 0,012)$$
$$b = (6,754 \pm 0,031)$$

This regression allows to infer how many agents are needed to complete the task below 1 hour :

$$N_{1hourAVG} \approx 1565$$

This first result is to be compared with the probability of finding a task in less than 1 hour as shown in Figure 1 :

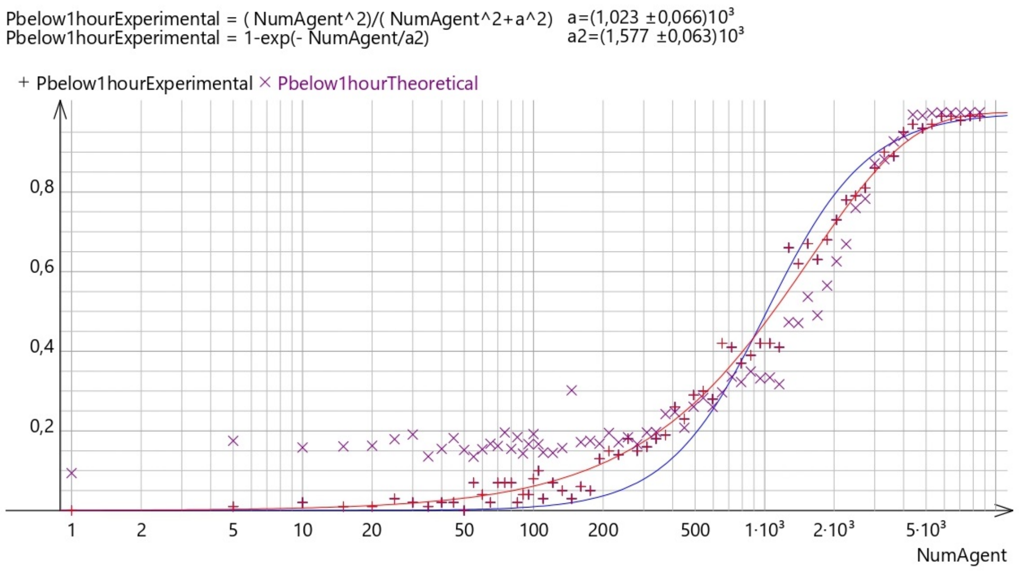


Figure 5: Probability of finding a task in less than 1 hour

Where :

- Pbelow1hourExperimental : is the measured probability of finding a task in time below 1 hour.
- Pbelow1hourTheoretical : is the theoretical probability of finding a task in time below 1 hour, if that probability were to be Gaussian, with mean and variance extracted from simulation results, for each number of Agents.
- Red regression : according model $P(t < 1h) = 1 - e^{\frac{-NumAgent}{a2}}$
- Blue regression : according model $P(t < 1h) = \frac{NumAgent^2}{NumAgent^2+a^2}$

Using the red regression (which fit better than the blue one), it can be found that for $N_{1hourAVG} \approx 1565$ the probability of completing the task in less than 1 hour is $P(t < 1h)(N_{1hourAVG}) = 0.63$. It shows that P(t<1h) is highly stochastic. If a 95 rate was required, one would need 4725 Agents. Those 4725 agents would complete a task with an average time of 1191 s.

In the situation where computation were to be slow (ex : without choice made in (Q1)), using the same regression could have been possible by running simulation up to 500 agents. If 500 agents is still to much, we could make a Regression of the threshold by running several simpler simulations. We could try to make that regression given the appropriate Hyper-parameters. A guess would be to use :

$$SpannedArea = \frac{2 * r * V * 3600 * \eta}{S}$$

Such that

$$treshold = f(SpannedArea)$$

Where η is the spatial efficiency if we do not use the trick mentioned in (Q1).

In the case of that simulation, the average time can be easily guessed by a regression as we did, even for high values of N. Using a Regression on the threshold would be thus over-killed. That approach is nonetheless valuable when simulation is not possible.

3 Appendix

```
#pragma once
#include <time.h>
#include <cstdlib>
#include <cmath>
#include <boost/geometry.hpp>
#include <vector>

/*! \file gameEntity.h
    \brief Define Crops and Agents CLASS






*/
#define DEFAULT_SPEED 10.0
#define DEFAULT_TR 5.0
#define BORDERSIZE 3162


typedef enum _border
{
    ok,
    N,
    S,
    E,
    O,
    NE,
    NO,
    SE,
    SO,
    UNDEF
}Border;


class WeedCrop
```

```
{
    private:
        double x;
        double y;
        long long int issuedDate;
        long long int obtainedDate;
        bool obtained;
    public:
        WeedCrop(long long int date);
        ~WeedCrop();
        static int TotalnumberOfCrops;
        double getX(void){return x;};
        double getY(void){return y;};
        void acquireCrop(long long int date){obtainedDate=date; obtained = true;};
        long long int getDateIssued(void){return issuedDate;};
        long long int getDateObtained(void){return obtainedDate;};
        bool getObtainedState(void){return obtained;}
        double getSearchRectangleBasePointX(double theta) {return x*cos(theta) +
↪   y*sin(theta);};
        double getSearchRectangleBasePointY(double theta) {return y*cos(theta) -
↪   x*sin(theta) ;};

};




class Agent
{
    private:
        double x;
        double y;
        double lastx;
        double lasty;
        double theta;
        double Tr;
        double v;
        int id;

    public:
        Agent(bool randompos,double v_input = DEFAULT_SPEED , double Tr_input = DEFAULT_TR);
        Agent(double x_input, double y_input, double v_input = DEFAULT_SPEED, double
↪   Tr_input = DEFAULT_TR );
        ~Agent();
        Border isPosOkay(double x, double y);
        static int TotalnumberOfAgent;
        void updatePos(double dt, double degRange =5.0);
        double getX(void){return x;};
        double getY(void){return y;};
        double getTr(void){return Tr;};
        double getVmeterPerSecond(void){ return v/3.6;};
        double getTheta(void){return theta;}
        double getSearchRectangleBasePointX() {return lastx*cos(theta) + lasty*sin(theta);};
        double getSearchRectangleBasePointY() {return lasty*cos(theta) - lastx*sin(theta) -
↪   Tr;};


};

#include "../include/gameEntity.h"
```

```
/*! \file gameEntity.h
    \brief Inplement Crops and Agents CLASS


*/

int Agent::TotalnumberOfAgent = 0;
int WeedCrop::TotalnumberOfCrops = 0;

WeedCrop::WeedCrop(long long int date)
{
    x = rand()%(BORDERSIZE-1) + (rand()%100)/100; //Random init
    y = rand()%(BORDERSIZE-1) + (rand()%100)/100;
    issuedDate = date; // Creation date to keep track of performances
    obtainedDate = -1;
    obtained = false;



    TotalnumberOfCrops++;
}

WeedCrop::~WeedCrop()
{

    TotalnumberOfCrops--;

}

Agent::Agent(bool randompos,double v_input , double Tr_input)
{
    if(randompos) //Random position init
    {
        x = rand()%(BORDERSIZE-1) + (rand()%100)/100;
        y = rand()%(BORDERSIZE-1) + (rand()%100)/100;
    }
    else
    {
        x=500.0;
        y=500.0;
    }


    theta = ((rand()%360)*M_PI)/(180.0);
    if(theta > M_PI)
    {
        theta -=2*M_PI;
    }
    if(theta < -M_PI)
    {
        theta +=2*M_PI;
    }


    TotalnumberOfAgent++;
    Tr = Tr_input;
    v = v_input;
    id = TotalnumberOfAgent;
    lastx = x;
    lasty = y;

}



Agent::Agent(double x_input, double y_input, double v_input , double Tr_input )
{

    if( isPosOkay(x_input,y_input)==Border::ok)
    {
        x = x_input;
        y = y_input;
    }
    else
    {
        x = rand()%(BORDERSIZE-1) + (rand()%100)/100;
        y = rand()%(BORDERSIZE-1) + (rand()%100)/100;
    }
    TotalnumberOfAgent++;
    Tr = std::fabs(Tr_input);
    v = std::fabs(v_input);
    id = TotalnumberOfAgent;
    lastx = x;
    lasty = y;

}

Border Agent::isPosOkay(double x, double y)
{
    if( x > 0.0 && x < BORDERSIZE && y > 0.0 && y < BORDERSIZE )
    {
        return Border::ok;
    }
    else if( x <0.0)
    {
        if(y > 0.0 && y < BORDERSIZE)
        {
            return Border::O;
        }
        if(y < 0.0)
        {
            return Border::SO;
        }
        if(y > BORDERSIZE)
        {
            return Border::NO;
        }

    }
    else if( x > BORDERSIZE)
    {
        if(y > 0.0 && y < BORDERSIZE)
        {
            return Border::E;
        }
        if(y < 0.0)
        {
            return Border::SE;
        }
        if(y > BORDERSIZE)
        {
            return Border::NE;
        }
    }
    else if( y < 0.0)
    {

        return Border::S;

    }
    else if( y > BORDERSIZE)
    {
        return Border::N;
    }
    return Border::UNDEF;
}

void Agent::updatePos(double dt,double degRange) //Update pos of one Agent
{
    lastx = x;
    lasty = y;
    int degRangeint = 1000*degRange; //Generating random angle with 10^-3 precision
    bool sgn = rand()%2; // Generating random sign
    int thetaRandint = rand()%degRangeint; //Generating random angle with 10^-3 precision
    double thetaRand = ((((double)(thetaRandint))*(M_PI))/(1000.0*180.0)); //Converting to
↪   RAD
    if(sgn == true)
    {
        thetaRand=-thetaRand; //Applying random sign
    }
    double thetaTemp = thetaRand + theta; // Calculating next Theta
    if(thetaTemp > M_PI)//Rescaling between [-PI,PI]
    {
        thetaTemp -=2*M_PI;
    }
    if(thetaTemp < -M_PI)
    {
        thetaTemp +=2*M_PI;
    }
    double xtemp =  x+(v/3.6)*dt*cos(thetaTemp); // Calculating next X
    double ytemp =  y+(v/3.6)*dt*sin(thetaTemp); // Calculating next Y
    if(isPosOkay(xtemp,ytemp)==Border::ok) //Checking for Border collision
    {
        theta = thetaTemp; // if no collision -> temp x, temp y, temp theta become next x
↪   next y next theta
        x = xtemp;
        y = ytemp;

    }
    else
    {
        theta += M_PI; //else if collision Theta +=PI
        if(theta > M_PI)
        {
            theta -=2*M_PI;
        }
        if(theta < -M_PI)
        {
            theta+=2*M_PI;
        }
        x =  x+(v/3.6)*dt*cos(theta); //Going Backward
        y =  y+(v/3.6)*dt*sin(theta);

    }
    if(isPosOkay(x,y)!=Border::ok) // Safety if something crazy happened
    {
        x = BORDERSIZE/2;
        y = BORDERSIZE/2;
        theta = 0.0;
    }



}

Agent::~Agent()
{

    TotalnumberOfAgent--;

}

#pragma once
#include "gameEntity.h"
#include <algorithm>


/*! \file grid.h
    \brief Define Grid class, wich contains all agents, crops,
    provide update fonctions, and utility for visualization


*/


#define WINDOW_SIZE 720
#define AGENTWEEDWIDHT 16
#define AGENTHEIGHT 9
#define WEEDHEIGHT 19

typedef struct _pos
{
    int x;
    int y;
}pos;

class Grid
{
    private:

        std::vector<WeedCrop> weedCrops;
        int numberOfAgents;
        int numberOfWeedCrop;
    public:
        std::vector<pos> agentPos;
        std::vector<Agent> agents;
        std::vector<pos> weedPos;
        std::vector<WeedCrop> obtainedWeedCrops;
        Grid(int numberOfAgents=1,int numberOfWeedCrop=1, double Vit= 10.0, double Tr =
↪   5.0);
        ~Grid();
        void updateAgents(long long int date, double dt, double degRange = 5.0);
        void ConvertPos();
};

#include "../include/grid.h"

/*! \file grid.cpp
    \brief Implement Grid class, wich contains all agents, crops,
    provide update fonctions, and utility for visualization


*/

Grid::Grid(int numberOfAgents,int numberOfWeedCrop,double Vit, double Tr)
{
    //INITIALIZATION
    for(int i = 0; i<numberOfAgents; i++)
    {
        agents.push_back(Agent(true,(double)Vit,(double)Tr));

    }
    for(int i = 0; i<numberOfWeedCrop; i++)
    {
        weedCrops.push_back(WeedCrop(0));
    }
    ConvertPos();//used for visualization
}

void Grid::ConvertPos() // Convert x/y position to picture position for visualization
{
    if(!weedPos.empty())//empty the vector wich contains previous position
        weedPos.clear();
    if(!agentPos.empty())
        agentPos.clear();
    for( Agent agent: agents)
    {
        double x = (double)((double)(agent.getX())*WINDOW_SIZE)/BORDERSIZE-AGENTWEEDWIDHT/2;
        double y =
↪   720-(double)((double)(agent.getY())*WINDOW_SIZE)/BORDERSIZE-AGENTHEIGHT/2;
        int xint = x;
        int yint = y;
        pos toAddPos  {xint,yint};
        toAddPos.x = std::min( std::max(toAddPos.x,0), WINDOW_SIZE); //ensuring position
↪   stay in window
        toAddPos.y = std::min( std::max(toAddPos.y,0), WINDOW_SIZE);
        agentPos.push_back(toAddPos);
    }
    for( WeedCrop weed: weedCrops)
    {
        double x = (double)(((double)weed.getX())*WINDOW_SIZE)/BORDERSIZE-AGENTWEEDWIDHT/2;
        double y = 720-(double)((double)(weed.getY())*WINDOW_SIZE)/BORDERSIZE-AGENTHEIGHT/2;
        int xint = x;
        int yint = y;
        pos toAddPos  {xint,yint};
        toAddPos.x = std::min( std::max(toAddPos.x,0), WINDOW_SIZE); //ensuring position
↪   stay in window
        toAddPos.y = std::min( std::max(toAddPos.y,0), WINDOW_SIZE);
        weedPos.push_back(toAddPos);
    }

}
```

```
void Grid::updateAgents(long long int date, double dt, double degRange ) //Perform one
↪  simulation step
{
    int toadd = 0; // Number of Crops to add at the end of the update
    for(auto it = std::begin(agents);it!=std::end(agents);it++)
    {
        it->updatePos(dt,degRange); // Update position of agent it according movement
↪  policy

        if( !weedCrops.empty())
        {

            for( auto it2 = std::begin(weedCrops); it2!=std::end(weedCrops); it2++) //Check
↪  if weedcrop it2 is near agent it1
            {
                    if( !weedCrops.empty())
                    {


                        double dist =
↪  sqrt(pow((it->getX()-it2->getX()),2)+pow((it->getY()-it2->getY()),2));
                        double xbasepoint = it->getSearchRectangleBasePointX();
                        double ybasepoint = it->getSearchRectangleBasePointY();
                        double theta = it->getTheta();
                        double cropx = it2->getSearchRectangleBasePointX(theta);
                        double cropy = it2->getSearchRectangleBasePointY(theta);
                        double v = it->getVmeterPerSecond();
                        double Tr = it->getTr();
                        bool checkRectangle = false;
                        if( cropx > xbasepoint && cropx < xbasepoint + dt*v )
                        {
                            if(cropy > ybasepoint && cropy < ybasepoint + Tr )
                            {
                                checkRectangle = true;
                            }
                        }
                        if( dist < Tr || checkRectangle ) // if dist < Tr => Remove Crops
↪  it2
                        {

                                it2->acquireCrop(date); //Register acquired date to
↪  calculate performances
                                obtainedWeedCrops.push_back(*it2);
                                weedCrops.erase(it2);
                                toadd++;
                                if(it2 != std::begin(weedCrops))
                                    it2--;
                                if(weedCrops.empty())
                                {
                                    break;
                                }



                        }
                    }
            }
        }

    }
    for(int i = 0; i< toadd; i++)
    {
        weedCrops.push_back(WeedCrop(date)); // Add as many crops as found
    }

}

Grid::~Grid()
{
}

#ifndef SDLDDRH
#define SDLDDRH

#include <cstdlib>
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <SDL2/SDL_ttf.h>

/*! \file sdlDDR.h
    \brief Define input structure generated in main by user


*/

typedef struct _mainSdlInput
{
    int N; //Number of Agents
    int Nc; // Number of crops
    double v; // Speed of Agents
    double Tr; // Detection Radius
    int timeMult; // (Visual) Simulation time multiplier
    bool keepRatioTime; // Option to Keep Visual simulation synchronized with
↪   realTime*timeMult
    bool renderImage; // Option to toggle Visualization, exit sim by closing window
    unsigned int numberOfCropsCollectedBeforeStop; //In case of no visualization, the number
↪   of Crops to obtained before stopping simulation
} mainSdlInput;



void mainsdl(mainSdlInput input);
void SDL_ExitWithError(const char *message);



#endif

#include "../include/sdlDDR.h"
#include "../include/gameEntity.h"
#include "../include/grid.h"
#include "time.h"
#include <iostream>


/*! \file sdlDDR.cpp
    \brief Perform simulation according to user inputs


*/
```

```
#define REFRESHTIME 16
#define MAX_SKIP 10


/*
int main(int argc, char** argv)
{
    mainsdl(0);
}
*/


void mainsdl(mainSdlInput input)
{

    srand(time(NULL)); //seed for random number generation
    Grid agrid(input.N,input.Nc,input.v,input.Tr);





    SDL_Window *window = NULL;
    SDL_Renderer *renderer = NULL;



    float dt = 1.0; // Simulation step
    int dt_int_ms = 1000; // Simulation step (ms)
    int delay_sim = (dt_int_ms)/input.timeMult;
    long long int simTime = 0; //Simulation time (ms)



    if(input.renderImage) //If user want visualization
    {
        float refreshtimef = REFRESHTIME;
        int refreshtime = refreshtimef;
        int ticks1;
        int ticks2;
        if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) != 0)
            SDL_ExitWithError("Initialisation SDL");
        IMG_Init(IMG_INIT_PNG);
        SDL_Surface *icon = IMG_Load("icon.png");
        SDL_Surface * imagebg = IMG_Load("backg.png");
        SDL_Surface * agent_surface = IMG_Load("saucer.png");
        SDL_Surface * weedcrop_surface = IMG_Load("weedcrop.png");





        //Création fenêtre
        if (SDL_CreateWindowAndRenderer(WINDOW_SIZE, WINDOW_SIZE, 0, &window, &renderer) !=
↪   0)
            SDL_ExitWithError("Creation fenetre et rendu echouee");



        SDL_SetWindowTitle(window, "TEK5010 - Assignement 1");
        SDL_SetWindowIcon(window, icon);
        SDL_Texture * texturebg = SDL_CreateTextureFromSurface(renderer, imagebg);
        SDL_Texture * textureAgent = SDL_CreateTextureFromSurface(renderer, agent_surface);
        SDL_Texture * textureWeedCrop = SDL_CreateTextureFromSurface(renderer,
↪   weedcrop_surface);






        int reftime_since_begin = SDL_GetTicks();
        int reftime = SDL_GetTicks();
        int reftime2 = SDL_GetTicks();
        SDL_bool program_launched = SDL_TRUE;
        SDL_RenderCopy(renderer, texturebg, NULL, NULL);
        int render = 0;
        bool nowait = false;



            //Lancement SDL


        while (program_launched)
        {
            ticks1 = SDL_GetTicks() +refreshtime;







            SDL_Event event;
            if (SDL_PollEvent(&event))
            {
                switch (event.type)
                {

                case SDL_QUIT:
                    program_launched = SDL_FALSE;
                    break;


                default:
                    break;
                }
            }

            reftime2 = SDL_GetTicks();
            if( reftime2-reftime > delay_sim)
            {

                agrid.updateAgents((simTime+dt_int_ms),dt);
                simTime+=dt_int_ms;
                agrid.ConvertPos();
                SDL_Rect workingRect {0,0,AGENTWEEDWIDHT,AGENTHEIGHT};
                if(input.keepRatioTime && render < MAX_SKIP && reftime2-reftime_since_begin
↪   > ((float)((float)delay_sim*((float)(simTime/dt_int_ms)))) + 4*delay_sim )
                {
                    render += 1;
                    std::cout << "Time steps :" <<  (reftime2-reftime_since_begin)/delay_sim
↪   << ", " << simTime/dt_int_ms << std::endl;

                }
                else if ( render >= MAX_SKIP || render==0)
                {

                    SDL_RenderCopy(renderer, texturebg, NULL, NULL);
                    for( auto it : agrid.agentPos)
                    {
                        workingRect.x = it.x;
                        workingRect.y = it.y;
                        SDL_RenderCopy(renderer,textureAgent,NULL,&workingRect);

                    }
                    workingRect = {0,0,AGENTWEEDWIDHT,WEEDHEIGHT};
                    for( auto it : agrid.weedPos)
                    {
                        workingRect.x = it.x;
                        workingRect.y = it.y;
                        SDL_RenderCopy(renderer,textureWeedCrop,NULL,&workingRect);

                    }
                    if(render > MAX_SKIP)
                    {
                        nowait = true;
                        render = 0;
                    }
                    else
                    {
                        render = 0;
                        nowait = false;
                    }


                }


                reftime = SDL_GetTicks();
                reftime2 = SDL_GetTicks();
            }







            ticks2 = SDL_GetTicks();



            if(ticks2 < ticks1 && render==0 && nowait==false)
            {
                if(reftime2-reftime < (delay_sim - (ticks1-ticks2)) )
                {
                    SDL_Delay(ticks1-ticks2);
                }
                else if ( input.keepRatioTime == false)
                {
                    SDL_Delay(ticks1-ticks2);
                }
            }








            if(render==0)
            {
                SDL_RenderPresent(renderer);
            }






        }
        SDL_DestroyRenderer(renderer);
        SDL_DestroyWindow(window);
        SDL_FreeSurface(icon);
        SDL_FreeSurface(agent_surface);
        SDL_FreeSurface(weedcrop_surface);

        IMG_Quit();

    }
    else // No visualization
    {
        while(agrid.obtainedWeedCrops.size() < input.numberOfCropsCollectedBeforeStop)
↪   //Wait to collect the specified number of crops
        {
            agrid.updateAgents((simTime+dt_int_ms),dt); //Update grid
            simTime+=dt_int_ms; // Increase simulation time

        }

    }

    //Show metrics
    std::cout<<"--------Simulation END--------\n"<<std::endl;

    // std::cout<<"Results : "<<std::endl;
    // std::cout<<"Simulation time : "<<simTime/1000.0<<" s"<<std::endl;
    // std::cout<<"NumberOfCropsFound : "<<agrid.obtainedWeedCrops.size()<<std::endl;
    long double avg = 0.0;
    long double variance = 0.0;
    int count_below_1_hour = 0;
    int numCollected = agrid.obtainedWeedCrops.size();
    if(agrid.obtainedWeedCrops.size() !=0)
    {
        for(auto it : agrid.obtainedWeedCrops)
        {
            long double t = (it.getDateObtained()-it.getDateIssued())/1000.0;
            if(t < 3600.0)
                count_below_1_hour++;
            avg+=t/numCollected;
        }
        std::cout<<std::endl;

        for(auto it : agrid.obtainedWeedCrops)
        {
            variance+=pow(avg-(it.getDateObtained()-it.getDateIssued())/1000.0,2);
        }
        variance = sqrt(variance/agrid.obtainedWeedCrops.size());
        std::cout<<"Nagent : "<<input.N<<std::endl;
        std::cout<<"Average Time for a crop to be found : "<<avg<<std::endl;
        std::cout<<"Associated Variance  : "<<variance<<std::endl;
        std::cout<<"Number of crops found below 1 hour "<<count_below_1_hour<<std::endl;

    }



    /*------------------------------------------------------------*/







}

void SDL_ExitWithError(const char *message)
{
    SDL_Log("ERREUR : %s > %s\n", message, SDL_GetError());
    SDL_Quit();
    exit(EXIT_FAILURE);
}


#include "../include/sdlDDR.h"
#include <iostream>
#include <vector>
#include <algorithm>


/*! \file main.cpp
    \brief Acquire user inputs and launch simulation


*/


mainSdlInput input = {1,1,5.0,10.0,1,true,true,100};


int nagentGenerator()
{
    static int i = 1;
    static int count = 4385;
    if(i++ > 20)
        count = 4385;
    count = count*1.1;
    return count;
}
```

```
int main(int argc, char* argv[]) {







    printf("\n");
    printf("*------------------------------------------------------------* \n");
    printf("Welcome To Particle Simulator - TEK5010 - Assignement 1 \n");
    printf("*------------------------------------------------------------* \n");
    printf(" \n");
    printf(" \n");
    printf(" \n");
    // printf("Enter number of Agents (int) : \n");
    // std::cin>>(input.N);
    printf("Enter number of Crops (int) : \n");
    std::cin>>(input.Nc);
    printf("\n \n Enter radius Tr (double) : \n");
    std::cin>>(input.Tr);
    printf("\n  Enter Agents'speed km/h (double) : \n");
    std::cin>>(input.v);
    printf("\n render image (bool) : \n");
    std::cin>>(input.renderImage);
    if( input.renderImage == false)
    {
        printf("Wait for how many collected Crop (int) ? \n\n");
        std::cin>>(input.numberOfCropsCollectedBeforeStop);
    }
    else
    {
        printf("\n \n Enter Simulation Time multiplier (int) : \n");
        std::cin>>(input.timeMult);
        printf("\n \n Skip frames if slower than realtime*time_multiplier (bool) : \n");
        std::cin>>(input.keepRatioTime);
    }














    /* ------------------------- */


            // SESSION SDL
   std::vector<int> tab (20);
   std::generate (tab.begin(), tab.end(), nagentGenerator);
   for(auto it : tab)
   {
        input.N = it;
        mainsdl(input);
   }
   /* ------------------------- */

    SDL_Quit();
        return EXIT_SUCCESS;
}
```