Predator/Prey Relationships

New Mexico Adventures
in
Supercomputing Challenge
Final Report
April 6, 2005

Team 57
Santa Fe High School

Team Member:
Taryn Flock

Teacher:
Anita Gerlach

Mentor:
Steve White

Table of Contents:

Executive Summary:

In this project, I attempted to gain insight into the nature of predator/prey relationships in respect to migration. The task was to compute, using a Monte Carlo based model, the development of a population of predators and preys when migration is allowed and to search for stable solutions.

Solutions of the predator/prey problem are generally unstable; however, Lokta and Voltera, separately, showed that stable solutions exist when migration is ignored. Does the problem, once modified to allow for migration, still have stable solutions? My results indicate a possible stable solution, which should be explored with a parallel implementation. The results also show that in the absence of migration this possible stable solution deteriorates rapidly, reflecting the importance of migration on the evolution of a population of predators and prey.

Introduction:

The problem of predator/prey relationships has many applications, not only in the wild world from where it comes, but in the interaction of reactants in chemistry as well. The predator/prey problem has been well researched when the spatial conditions of the problem are ignored. However, the behavior of this problem, when modified to allow for migration by adding a spatial grid, is not well described. In order to tie my interest in learning Monte Carlo with an idea of my mentor, we decided to explore the behavior of predator/prey relationships in respect to migration. The goal was to find stable oscillating solutions in the populations of predators and prey.

Methods:

Any physical system can be described in two ways, the first is to find governing equations that represent the system, the second is to observe the system and note the relative frequency with which various events occur. One can then mimic that behavior by randomly sampling these occurrences in a computation at the observed relative frequency. This class of modeling techniques is generally known as the Monte Carlo method.

Extrapolating from the Lokta-Voltera model (see appendix A), we developed the following rules for the system:

1. Geometry, extent and boundary conditions

1.1 Temporal: 150 cycles. (A cycle represents a day.)

1.2 Spatial: 10 by 10 distance units (miles). The problem will evolve within this domain, thus there are 100 cells ($100mi^2$) in the problem.

1.2.1 A trajectory of either predator or prey that would attempt to leave this extent is reflected off the boundary, back into the problem as if the boundary were water.

1.2.2 The predator/prey evolution is localized to the populations within each cell. The rules for the evolution are described below.

2. Initial conditions

2.1 Predator population: 500 predators are distributed uniformly across the spatial grid. They are given isotropic initial directions. Their initial velocity is 0.1 [cells/ cycle].

2.2 Prey population: 500 preys are distributed uniformly across the spatial grid. They are given isotropic initial directions. Their initial velocity is 0.1 [cells/cycle].

3. Predator evolution constraints

3.1 Starvation: a predator will be removed from the simulation if it has not eaten a prey within 10 consecutive cycles.
3.2 The probability of eating a prey is 0.55 [per cycle] if the local prey population is non-zero.
3.3 Reproduction: the probability that one new predator is inserted into the simulation is 0.48 [per meal].
3.4 Trajectory: the probability that a predator will change direction, isotropically, is 0.3 [per cycle]. The predator's speed will be 0.1*(1.05^days since last meal) (i.e., it will range further the hungrier it gets).

4. Prey evolution constraints

4.1 Preys are removed from the simulation as they are consumed by predators within the local cell of the predator.
4.2 Reproduction: the probability that prey reproduce is 0.0632 [per cycle]. Six (6) new preys are inserted in the simulation upon a reproduction.
4.3 Trajectory: the probability that a prey will change direction isotropically is 0.3 [per cycle]. The prey's speed is based on the population concentration within the cell using the formula 0.1*[10^(population in the cell/total population)] (i.e., prey will move faster the more crowded they become).

The simulation initializes the system by creating a linked list of creatures randomly assigned (with 50/50 probability) to predator or prey. Preys have initial weight six, and predators, weight one, as described in the rules. Each is given a random location and direction and a velocity of 0.1. The program counts the population within each cell.

The simulation then executes one day, in accordance with the rules, keeping track of the death toll but not removing the deceased prey from the list. The next step is to "cull"—remove the proper number of prey from the list—and to adjust the velocity of the prey in each cell in accordance with the new population figures. The predator velocity is also updated at this time. During this process, weight control is used (on prey only) to help free memory of insignificant list members, and to divide large list members (25% of cell population) into multiple list members to better model real behaviors.

Results are then displayed.

Results:

The model gives a solution that oscillates within the range examined. A second trial shows a similar oscillation.

Figure 1:

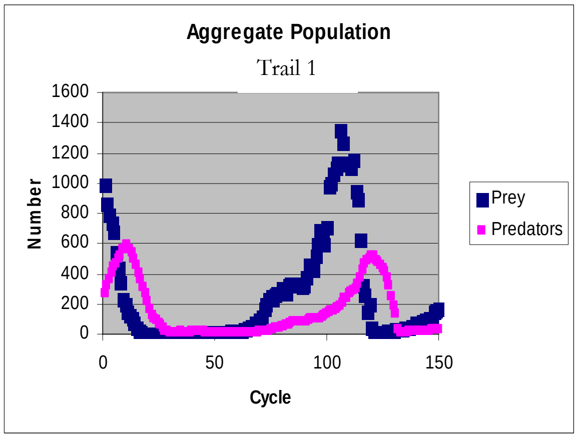


Figure 2:

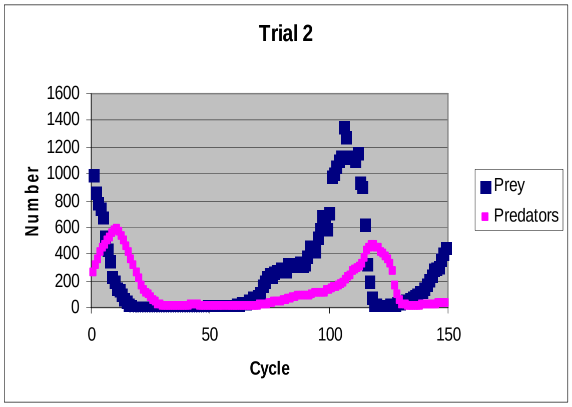
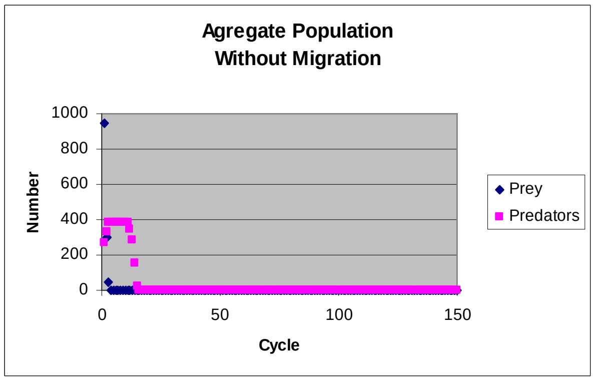


Figure 3:



Running the program again but eliminating the grid so that there is only one cell, and hence no migration, causes the population to die off almost immediately.

Conclusions:

The oscillation shows that my model may be stable, which means that it deserves further exploration to see if it may be used to represent the real-world interaction within a population. Also the rapid elimination of the population in the trial that excluded migration, shows that migration has a significant effect on the problem and is deserving of the time and effort put into it so far, and is deserving of further study as well.

Recommendations:

The next step is to carry the simulation over a larger time interval to see if the oscillation continues. Although, while in the process of finding this solution I tried many constants, trial of further constants is recommended. To test the reality of the problem, however, the simulation needs to be compared to the reality of different animals or chemical reactions, to see where in the real world it may be applied. I hope to explore the stability of my possible solution with a parallel implementation in the coming summer.

Acknowledgements:

My mentor, Steve White, and teacher, Ms. Gerlach, were an invaluable source of motivation and aid.

Appendix A:

The Lokta-Voltera Model:

$$R(n+1) = R(n) + a*R(n) - b*R(n)*F(n)$$
$$F(n+1) = F(n) + e*b*R(n)*F(n) - c*F(n)$$

Where the parameters are defined by:

- a is the natural growth rate of rabbits in the absence of predation,
- c is the natural death rate of foxes in the absence of food (rabbits),
- b is the death rate per encounter of rabbits due to predation,
- e is the efficiency of turning predated rabbits into foxes. [1]

[1] From http://www.stolaf.edu/people/mckelvey/envision.dir/nonDE.lotka-volt.html

Appendix B: The Program

The Header file:

```
/* Predator/ Prey Library */

struct R2 {                  // Vector for 2-D Location and direction
            float x, y;
} ;

struct cx {                  // cell index in 2-D grid
            int i, j;
} ;

#define PRED 0
#define PREY 1

struct pxy {            // used for both predator and prey
             struct R2 xy_l;       // location
             struct R2 xy_d;       // direction
             double xy_wt;         // weight
             double xy_v;          // velocity
             double xy_t;          // time
             struct pxy *xy_next;  // pointer to next linked list element
             char xy_tag;          // PRED or PREY
} ;

#define NI 10               // number of cells in x direction
#define NJ 10               // number of cells in y direction
#define NX 200              // initial number of predators
#define NY 200              // initial number of prey
#define NCYCLES 150                              // number of cycles (time) for simulation

#define STARVE 10           // cycles without a meal before predator dies
#define WT0  6              // number of progeny/prey/cycle
#define VEL0 0.1            // intial velocity, in cells/cycle
#define VELY 10             // increase prey velocity with cell prey population
#define VELX 1.05           // increase predator velocity as hunger increases

#define PEAT 0.55           // number eaten/prey/cycle
#define PXREP .48           // predator reproduction probability/predator/meal
#define PDIR 0.3            // probability to change direction (isotropically)
#define PYREP .0632         // prey reproduction probability /prey/cycle
#define wc1 0.25            // weight control: split
#define wc2 0.01            // weight control: roulette
```

The Program:

```
/* ppxy.c
* ----------------------
* Predator Prey Problem:  A Monte Carlo Simulation including Migration
*/

#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>
#include "pxy.h"

void initialize ();
void process ();
void cull ();
void viewem (void);
double urand (void);
void iso (struct pxy *);
void migrate (struct pxy *);
double db (struct pxy *);
int cell (struct pxy *);

struct pxy *xyhead;
float *lpop, *dpop, *mpop;

#define MIN(a,b) ((a)< (b) ? (a):(b));
#define MAX(a,b) ((a)> (b) ? (a):(b));

void main () {
        int n = NCYCLES;

        //allocate memory for global variables
        lpop = (float *) calloc (NI*NJ, sizeof (float));
        dpop = (float *) calloc (NI*NJ, sizeof (float));
        mpop = (float *) calloc (NI*NJ, sizeof (float));

        //Begin application
        randomize();  //starts the random number generator with a new random seed
        initialize (); //creates linked list of creatures
        viewem ();     //displays results to screen

        //Begin Simulation in Ernest
        while (n--) {
                process ();
                cull ();
                viewem ();
        }
}

/* Urand()  takes no input returns a uniformly distributed random number
*       between zero and one
*/

double urand () {
        return (1.0*rand ()/RAND_MAX);
}

/*initialize()  takes no input, creates a linked list of predators and preys (in
*           random order) and initializes the values of type, location,
*           velocity, speed, direction, weight and type for the creatures
*/

void initialize () {
         int m, n, i;
```

```
            int nxy = NX + NY;          //number pred & prey
            double px = 1.0*NX/nxy;     // Probability initial item is a predator
            struct pxy *p;


            for (i=0; i<NI*NJ; i++) {    //Set all global variables to zero
                        lpop[i] = 0;
                        dpop[i] = 0;
                        mpop[i] = 0;
                        }

            for (n = 0; n < nxy; n++) {  //Create structure, link to top of list
                        p = (struct pxy *)calloc (1, sizeof (struct pxy));
                        p->xy_next = xyhead;
                        xyhead = p;
                                                                                                        //
initialize scalars
                        p->xy_t = 0;
                        p->xy_v = VEL0;
                                                                                                        //
initialize loc/ dir
                        p-> xy_l.x = NI*urand ();
                        p-> xy_l.y = NJ*urand ();
                        iso (p);

                        if (urand()< px) {          //set as predator or prey and assign weight
                                    p->xy_tag = PRED;
                                    p->xy_wt = 1.0;
                        } else {
                                    p-> xy_tag = PREY;
                                    m = cell (p);
                                    p-> xy_wt = WT0;
                                    lpop[m] += WT0;          //accumulate local population
                        }
            }
}

/* Viewem ()  takes no input and prints the current population, by cell, to the
*           screen
*/

void viewem () {
            struct pxy *p;
            int *xp = calloc (NI*NJ, sizeof (int));         //predator population by cell
            double *tb = calloc (NI*NJ, sizeof (double));   //average time since last fed
            double *xv = calloc (NI*NJ, sizeof (double));   //average pred speed
            double *yv = calloc (NI*NJ, sizeof (double));   //average prey speed
            int i, j, m;

            for (p=xyhead; p; p=p->xy_next) { //accumulate totals over all list members
                        m = cell (p);
                        if (p->xy_tag==PRED) {
                        tb[m] += p->xy_t;
                        xv[m] += p->xy_v;
                        xp [m] ++;
                        } else {
```

```
                        yv[m] += p->xy_v;
                        }
            }
            for (i=0; i<NI; i++) {              //Prints Results to screen
                        for (j = 0; j < NJ; j++) {
                                    m = j*NI +i;
                                    if (xp[m]) {
                                                tb[m] /= xp[m];
                                                xv[m] /= xp[m];
                                    }
                                    if (lpop[m]) yv[m] /= lpop[m];
                                    printf ("cell:%d %d xpop:%d tf:%f xv:%f   lpop:%f yv:%f\n",
                                                                        i, j, xp[m], tb[m], xv[m], lpop [m],
yv[m]);
                        }
            }
            free (xp); free (tb); free (xv); free (yv);
}

/* process()  takes no input and exectutes the process of a day, accumulates
*           death toll rather than killing off the prey as they are eaten
*/

void process (){
            struct pxy *p, *pp, **q;
            int i, m, m2;
            q = &xyhead;

            while (p = *q) {        //execute day over all list members
                        m = cell (p);
                        if (p-> xy_tag == PRED) {
                                    if (p->xy_t++ > STARVE) {           //predator starves
                                                *q = p->xy_next;
                                                free (p);
                                    } else {
                                                if (lpop[m] + dpop[m] + mpop[m]> 0) {
                                                            if (urand () < PEAT ){          //predator hunts
                                                                        p->xy_t = 0;
                                                                        dpop[m] -= WT0*urand();     //prey death recorded
                                                                        if (urand() < PXREP) {      //predator reproduces
                                                                                                //Child list member created
                                                                          pp = (struct pxy* )calloc (1, sizeof (struct pxy));
                                                                          pp->xy_l.x = p-> xy_l.x;     //same position as parent
                                                                          pp->xy_l.y = p-> xy_l.y;
                                                                          iso (p);                   //new direction
                                                                        pp->xy_t = 0; pp->xy_v = VEL0; pp->xy_wt = 1;
                                                                                    pp->xy_next = xyhead;
                                                                                    xyhead = pp;
                                                                        }
                                                            }
                                                }
                                                migrate (p);
                                                q = &p->xy_next;
                                    }
                        } else {
                                    if (urand () < PYREP) p->xy_wt *= (1.0 + WT0*urand ()); //prey reprod.
```

```
                migrate (p);            //prey migrates
                m2 = cell(p);
                if (m!= m2) {           //migration between cells recorded
                    mpop[m]-= p->xy_wt;
                    mpop[m2] += p->xy_wt;
                    }
                q = &p->xy_next;
            }
        }
    for (i=0; i<NI*NJ; i++)         mpop[i] = 0;  //reset migrant population
    }

    /* Cull ()  takes no imput, other that the global variables, and updates the
    *       the list to reflect the occurances in process ()
    */

    void cull () {
        struct pxy *p, *pp, **q;
        int i, m, n;
        double w, t, td, tl;
        int mx=0;
        int nc = NI*NJ;
        FILE *outfile;
/* To save data to be plotted to a different file change name within the quotes*/
        outfile= fopen ("clean.txt", "a+");
        if (outfile==NULL) printf ("error, bad file");

        for (i=0; i<nc; i++) lpop[i] = 0; //reset lpop
        n = mx;
        for (p = xyhead; p; p = p-> xy_next) {
            n++;            //count number of list members
            if (p-> xy_tag == PREY) {
                m = cell (p);
                if ((w = dpop[m]) < 0.0) { //if cell has diminished population
                    t = p->xy_wt;       //reduce the weight of prey list member
                    t = MIN (t, -w);    //in cell
                    p->xy_wt -= t;
                    dpop[m] += t;
                }
                lpop[m] += p-> xy_wt;
            } else mx++;            //count number of predators
        }
        td = tl = 0.0;                          //zero population counters
        for (i=0; i<nc; i++) {
            td += dpop[i];      // total discrepant diminished population
            tl += lpop[i];      // total prey population
            dpop[i]= 0.0;
        }

        /*Display results of cull, and save agregates to file*/
        printf ("cull: #:%d preds:%d tdd:%f preys:%f \n", n, mx, td, tl);
  fprintf (outfile, "%d     %f\n", mx, tl);
        fclose (outfile);

        /*Update list to reflect population changes*/
```

```
                for (q = &xyhead; p = *q;) {
                        m = cell (p);
                        if (p->xy_tag == PREY) {

                                        /*Implement Weight Control*/
                                if (p-> xy_wt < wc2*lpop[m]) {  // if prey weight is small relative
                                        lpop[m] -= p->xy_wt;          //to local population remove it
                                        *q = p-> xy_next;            // from the list
                                        free (p);
                                        } else {
                                        if (p->xy_wt > wc1*lpop [m]) { //if prey weight is large
                                                p->xy_wt *= .5;            //relative to the local
population

                        //split it into two list members
                                                pp = (struct pxy *) calloc (1, sizeof( struct pxy));
                                                pp->xy_l.x= p->xy_l.x;    pp->xy_l.y= p->xy_l.y;
                                                pp->xy_t = p->xy_t; pp-> xy_v = p->xy_v;
                                                pp->xy_wt = p->xy_wt; pp->xy_tag = PREY;
                                                iso (pp);      //give new creature a new direction
                                                pp-> xy_next = xyhead;
                                                xyhead = pp;
                                        }

                                 if (tl) {            //increase speed proportional relative
                                                      //to local population
                                                p->xy_v = VEL0*pow(VELY, (lpop[m]/tl));
                                 }
                                        q = &p->xy_next;
                                }
                        } else {
                                p->xy_v = VEL0*pow (VELX, 1.0* p->xy_t); //increase predator
                                                                            //speed relative to hunger
                                q = &p->xy_next;
                        }
                }
        }

        /* iso (struct pxy *p) given a pointer to a list member isotropically selects
        *                   a new direction for that list member.
        */

        void iso (struct pxy *p ) {
                double th = 2*M_PI*urand ();  //sample isotropic direction
                p-> xy_d.x = cos (th);        //assign direction
                p-> xy_d.y = sin (th);
        }

        /* migrate (struct pxy *p) given a pointer to a list member it updates the
        *                       location of the list member in accordance with
        *                       current velocity.
        */

        void migrate (struct pxy *p) {
                 double d, v, b, test;
                 double wx = p->xy_d.x;
                 double wy = p->xy_d.y;
                 double x = p->xy_l.x;
                 double y = p->xy_l.y;

                 for (v = p->xy_v; v>0; v-= d)  { //travel distance for one cycle at the
                                                   //speed 'v'; if one encounters the
                                                   //problem boundary first, reflect back
                                                   //in to the problem and repeat until the
                                                   //cycle's distance is met

                 b = db(p);                  //find distance to problem boundary
                 d = MIN (b, v);
                 x += d*wx;                  // advance list member
                 test =  MAX (0, x);
                 x = MIN ( test, NI );
                 y += d*wy;
                 test = MAX (0, y)
                 y = MIN (test , NJ );

                 if (x==0 || x==NI) wx = -wx;     //reflect off of top/bottom boundary
                 if (y==0 || y==NJ) wy = -wy;     //reflect off of left or right
                 if (urand ()<PDIR) iso (p);      //update direction
                 else {
                        p->xy_d.x = wx;
                        p->xy_d.y = wy;
                 }
        }
        p->xy_l.x = x;
        p->xy_l.y = y;
}

/* db (struct pxy *p)  given a pointer to a list member finds the distance to boundary for the
*       list member.  Returns distance to boundary.
*/

double db (struct pxy *p ) {
        double d = 1e10;
        double w, s;

        if (w = p->xy_d.x)  {                   //if not horizontal, find smallest positive
                 s = (NI - p->xy_l.x)/w;    //distance to top or bottom boundary
                 if (s > 0) d = MIN (d, s);
                 s = (0 - p->xy_l.x)/w;
                 if (s > 0) d = MIN (d,s);
        }
        if (w = p->xy_d.y)  {                   //if not vertical, find smallest positive
                 s = (NJ - p->xy_l.y)/w;    //distance to right or left boundary
                 if (s > 0) d = MIN (d, s);
                 s = (0 - p->xy_l.y)/w;
                 if (s > 0) d = MIN (d,s);
        }
        if (d==1e10) {              //neither: must be on the problem boundary
                        d=1e-3;             //give it a nudge, because otherwise

        // it causes problems
        }
        return (d);
}

/* Cell(struct psy p*) given a pointer to a list member finds the cell of
*               location of the list member.
*/

int cell (struct pxy *p) {
        int i = p->xy_l.x;   //read in x coordinate
        int j = p->xy_l.y;   //read in y coordinate

        i = MAX (0, i); i = MIN (i, NI-1);   //convert to cell
        j = MAX (0, j); j = MIN (j, NJ-1);
        return (j*NI + i);
}
```

Bibliography of Sources and Softwear:

Borland Turbo C++. Version 4.5

Lokta Volterra Model from:
http://www.stolaf.edu/people/mckelvey/envision.dir/nonDE.lotka-volt.html

Monte Carlo Method:
Carter, Cashwell. Particle-Transport Simulation with the Monte Carlo Method.

C Programming Review:
Roberts. The Art and Science of C.