Introduction

We have been asked by a group of paleontologists to model the hunting and evasion strategies of the predator *Velociraptor mongoliensis* as, alone or with a friend, he pursues the prey species *Thescelosaurus neglectus*; both extinct dinosaurs. The predator is 6/5ths faster than the prey, but has a 3-times larger turning radius at top speed. The chase continues for only 15 seconds, whereupon the predator must stop because of a lactic acid buildup in his muscles.

We devised two different models for this pursuit— a purely mathematical model, and a computer model. Both models use the following assumptions:

1.) All animals are represented as points at their respective centers of mass.
2.) The simulations and chase both begin when the predator, slowly sneaking up on the prey, is spotted by the prey. The distance between the animals at this instant is [hi].
3.) The chase lasts a maximum of 15 seconds, or until the prey is killed.
4.) No animal may travel at more than his maximum speed
5.) Each time the prey's distance to the predator is at a local minimum, the prey takes a chance of being killed, p(x).

Introduction to the Mathematical Model

The mathematical model makes the additional assumption that it takes a negligible amount of time to start and stop moving, but that the turning radius consideration limits the maximum angular velocity the animals are able to sustain. This assumption has the advantage of making the analysis possible without a computer, but neglects acceleration. This model is also only capable of analyzing one predator, one prey situations, but for those situations in which it works, it works quite quickly and well.

Using the mathematical model, we analyzed two predator strategies and one prey strategy: the prey strategy is to run away until the predator gets closer than some distance [hb], then make a sharp turn to the left or right. The prey can just squeak around the predator, whose larger turn radius makes him lose a few feet. The predator strategies considered were a hungry predator, who head straight for the current position of the prey, and the maximum turn predator, who uses his knowledge about how the prey reacts to make his turn sharper than the hungry predator.

Introduction to the Computer Model

In the computer model, we start with the assumption that the animals have an ability to accelerate in the direction of their choice. We then stick with a physics-based model of velocity and position, and use an iterative algorithm to figure out where predators and prey go, and which survive. By running several hundred thousand 15-second scenarios, we were able to determine how survival rates change depending on the predator strategy, the prey strategy, the initial separation [hi], the elevation and terrain, animal reaction times, etc.

The computer model supports N predators chasing M prey across any given terrain, and takes into account acceleration/deacceleration, reaction times, two predator strategies, and three prey strategies. Written in C++, and running on a PowerPC 604-based 132 mHz workstation, the program can calculate six hundred 15-second long scenarios using a 1/1000 of a second timestep in one minute. This enabled us to quickly and easily experiment with different strategies, and get statistically meaningful results.

The Velociraptor Problem *or* Lunch on the Run

We have been asked by a group of paleontologists to model the hunting and evasion strategies of the predator *Velociraptor mongoliensis* as, alone or with a friend, he pursues the prey species *Thescelosaurus neglectus*; both extinct dinosaurs. The predator is faster than the prey, but has a larger turning radius at top speed. The chase can continue for only 15 seconds.

Using both mathematical and computer models, we determined how the following factors affect survival rates:

1.) Predator Strategy
2.) Prey Strategy
3.) Reaction Time
4.) Elevation/Terrain
5.) Distance between the animals when the chase starts, h_i.

Since our models are able to produce probability of survival rates, we were able to graphically find which of the above factors most influences survival. We discovered that, despite quite different assumptions in our mathematical and computational models, both models agreed that the single most important factor in determining survival rates is the initial separation, h_i.

This is because our most successful prey strategy was to flee the predator directly, making a sharp turn at the critical distance h_b (1.619 m) from the predator. By utilizing its smaller turn radius in this fashion, the prey can get slightly ahead of the predator. The predator rapidly closes this distance, but then the prey repeats the same trick. Each time he does so, however, he takes another chance that the predator will kill him. Hence the number of breakaway turns he has to make is the most important factor in determining his chances for survival.

Various hunting strategies are also considered, as well as the two-predator case (whereupon sharply veering is less effective because a second predator, following the first, can make the kill). Nonetheless; because h_i is so important, our advice is this:

Predator, be stealthy. Prey, be vigilant.

The Computer Simulation Model:

Our simulator is physics-based, modeling each animal as a point at its center of mass, which is capable of choosing a direction in which to accelerate. This model outputs smooth curves representing the path of the center of motion of each animal. It does so by iteratively solving these vector differential equations using Euler's method (outlined below):

1.) $$\frac{d\vec{P}}{dt} = \vec{V} \qquad \frac{d\vec{V}}{dt} = \vec{A}$$

(where P,V, and A are the position, velocity, and acceleration as functions of time)

The simulation cycle begins by determining the optimal direction for the animal to accelerate based on the animal's strategy. The animal then accelerates at his maximum acceleration in that direction.

Values for this upper bound on acceleration are parameters to the simulation, derived from the minimum turn radius at top speed given for each animal in the problem description. We can work this out with the central acceleration formula, $a=v^2/r$.

The acceleration vector is then added to the local elevation gradient multiplied by the acceleration of gravity, making it easier to go down a hill than up. The elevation gradient is determined from a bi-linearly interpolated elevation grid, which is read in from an elevation file.

Once the animal has decided on a direction to accelerate in, the simulation applies one step of Euler's method to update his position and velocity vectors. This method of solving differential equations works by noting that the first two terms of the Taylor series expansion of a function of time depend only on the present condition of the system.

2.) $$f(t+h)=f(t)+h*f'(t)$$

Hence, if we know an animal's position, velocity, and acceleration at some time t, we can figure out a first-order approximation for where the animal will be at time $t+\Delta t$ by substituting equation 1.) into 2.):

$$\vec{P}(t+\Delta t)=\vec{P}(t)+\Delta t*\vec{V}(t)$$

$$\vec{V}(t+\Delta t)=\vec{V}(t)+\Delta t*\vec{A}(t)$$

In this way, using a Δt of 1/1000 of a second, our simulation works out, given the initial conditions and an acceleration vector, the velocity and position of each animal at any instant.

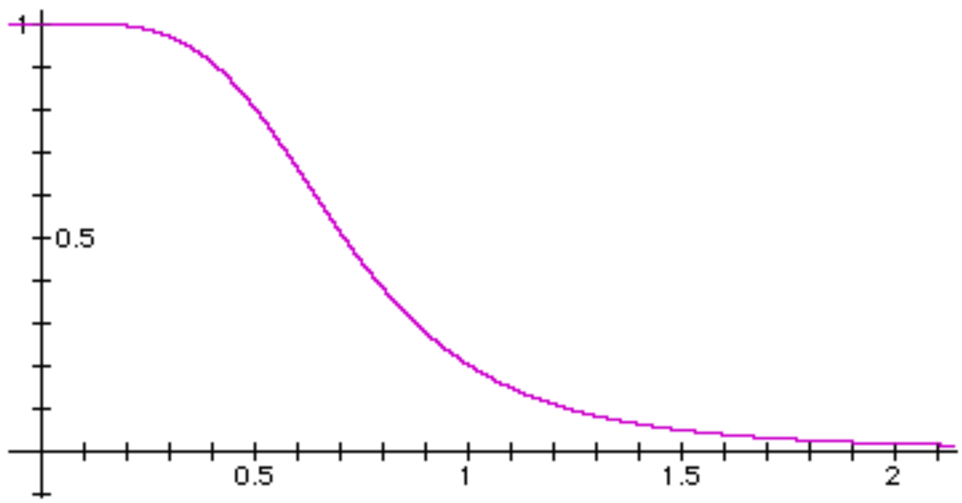
This simulation method is quite general, currently modeling several hunting and evasion strategies, elevation, reaction times, numeric or graphical output, and N predators vs. M prey. It is written in C++, and the entire program (10 pages) appears in appendix [C].

Life and Death in the Computer:

In the computer model, each time the prey passes near the predator, a probability of death function is evaluated. The probability of death given closest distance formula we chose is:

$$P(x)=\frac{1}{1+4*x^4}$$

This probability function looks like:



This function was chosen because the raptor's head and upper body is small and lightweight, while the main killing machinery are attached to the raptor's feet in the form of a large, curved hook. Hence it was assumed that the raptor would have to get his center of mass one leg-length (0.5 m) away from his prey's center of mass in order to substantially ensure a kill, but there remains a reasonable probability of a kill farther away if the predator uses his head or arms, or if the prey's tail is caught.

Since there is always some chance of a kill, at any distance (due to events such as the prey tripping, or breaking his leg, etc. which are not otherwise addressed), the probability function is asymptotic with respect to the x axis.

Simulated Hunting and Evasion Strategies:

In the simulation, each animal has perfect awareness of its own position and velocity, but only knows other animal's position and velocity one reaction time ago (e.g. 20 milliseconds).

Visually, this assumption is less justifiable for the prey than for the predator, since the prey would have to look backwards to see the predator, while the predator keeps his gaze fixed on the prey. But if we consider auditory perception as well, this "delayed telepathic knowledge" is fairly reasonable— the predator both sees and hears the prey, and the prey hears the predator's footsteps behind him.

The interesting question is: if you have some idea of where your prey or predator is, then how do you respond? Various modeled strategies for both predator and prey are discussed next.

Modeled Hunting Strategies:

Every orientation of two animals can be simulated by starting the predator at the origin and the prey a distance hi along the x-axis, where hi varies in this simulation from 15m to 50m.

The Hungry Hunter:

A hungry hunter heads straight for the current position of the closest prey. This hunter is subject only to the constraints of his knowledge about the prey's position and his own ability to get there. This is the only strategy modeled in the mathematical model, and has been rigorously analyzed there.

The Smart Hunter:

A smart hunter determines the point where he can intercept the closest prey, and heads straight for that point. A detailed derivation of the quadratic equation the smart hunter solves to determine the intercept point is given in Appendix [A].

Modeled Evasion Strategies (with 1-predator graphical results):

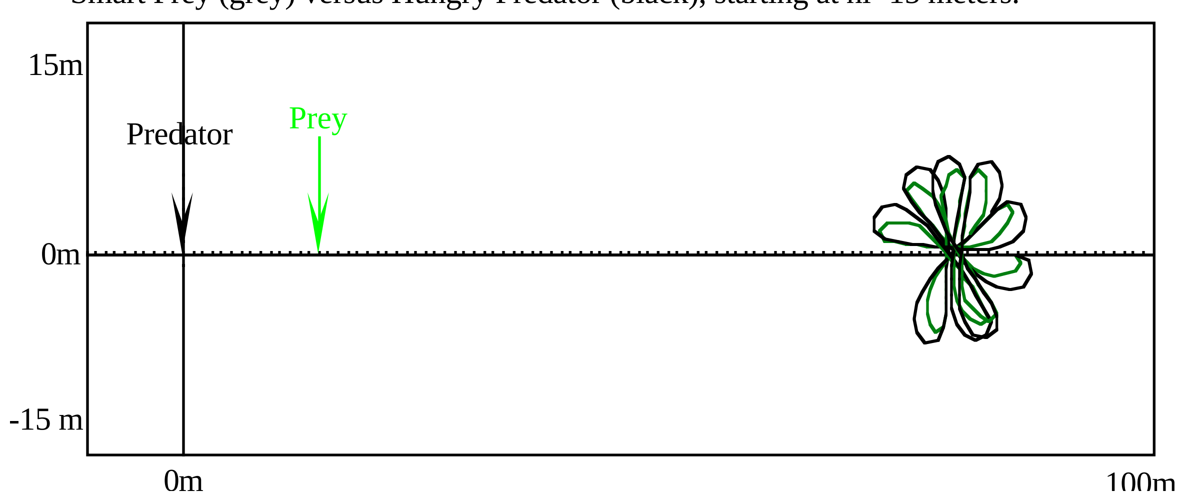
The Frightened Prey:

Frightened prey flee straight away from the nearest predator. These prey will always die if the predator can close the distance between them before his 15 seconds are up. At a closing rate of 2.7777 m/s, the prey will always be overtaken and die if hi (the initial separation of the predator and prey) is less than 41.6666 meters.

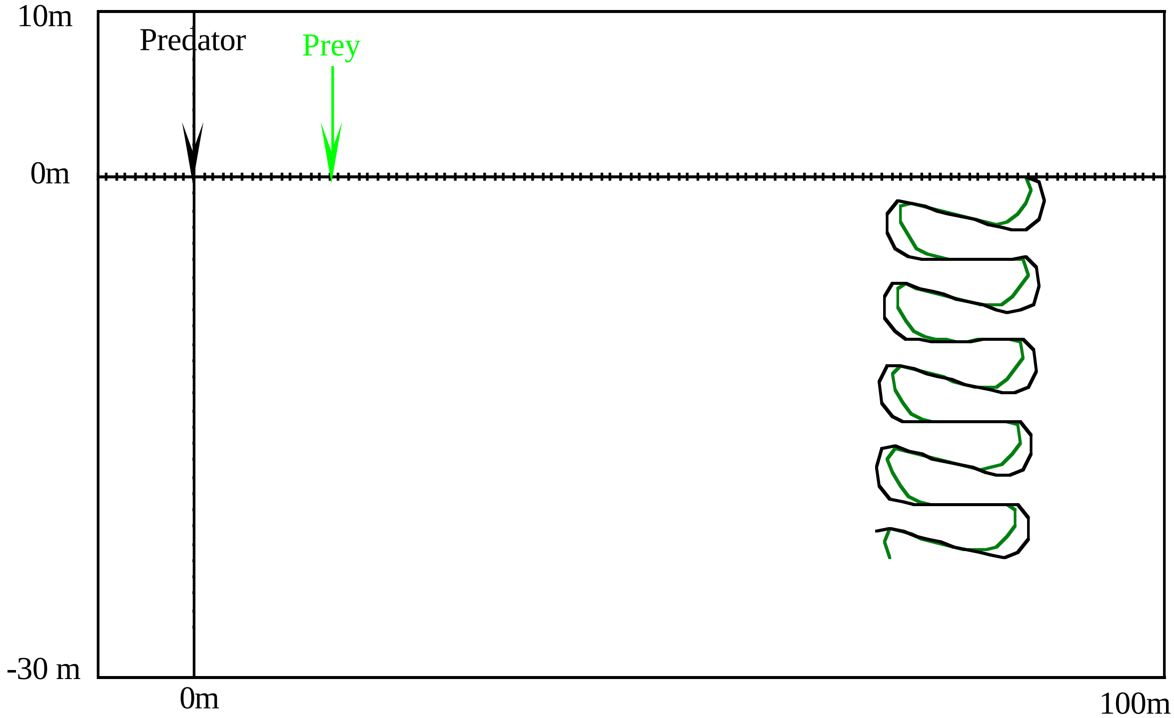
The Smart Prey:

When the nearest predator is far away, smart prey act like frightened prey and flee straight away. But when the predator closes to a critical distance hb (1.619 m— see Appendix [B] for derivation), the prey will dart either to the left or to the right. By using his much smaller turn radius, the prey buys some distance, which is again closed by the predator, whereupon the prey can dart again.

For the two-animal case, there are two possibilities for what this looks like: smart prey versus hungry predator, and smart prey versus smart predator.
Smart Prey (grey) versus Hungry Predator (black), starting at hi=15 meters.



Smart Prey vs. Smart Predator, starting at hi=15 meters:
(once again, the prey makes it).



The Gradient Prey:

Gradient prey are the same as smart prey when the nearest predator is very close (less than hb) or when there is only one predator. When there are two or more predators, the gradient prey runs in the direction of least danger— i.e. along the gradient of the danger function. If the danger from each predator decreases with the inverse square of its distance, and the danger from each predator is added to produce the danger function, then the danger gradient can be computed by adding the gradient of the danger from each predator. I.E.

$$\nabla \sum \frac{1}{d_i^2(x,y)} = \sum \nabla \frac{1}{d_i^2(x,y)}$$

This can be quickly and easily done in the simulation. The gradient prey is only different from the smart prey when there are two predators, so their graphs and analysis come in the next section.

The 2-Predator Situation:

After some experimentation, we've determined that the best strategy for the predators is for the second predator to follow the first one (by about 8 meters), since our prey's strategies hinge around getting behind the first predator.

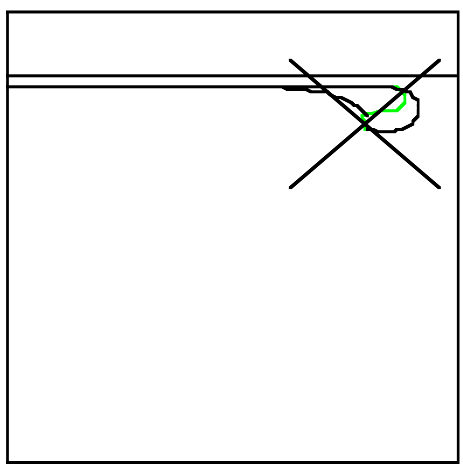
Frightened Prey:

Because frightened prey always die unless they detect the predator(s) more than 41.6666 meters away, there is no advantage to be had in chasing them with 2 predators.

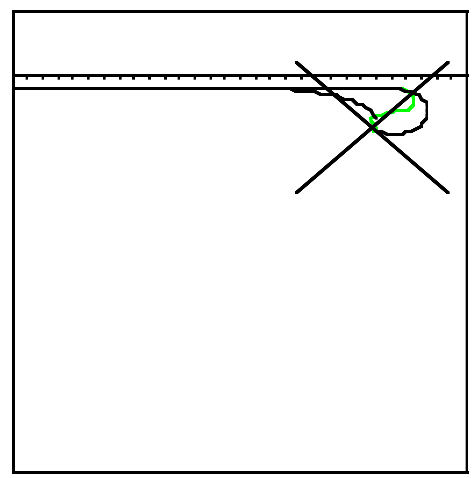
(plots from 60 to 90 m X, 5 to -30 m Y, predator 1 at -8 m, predator 2 at 0m, prey at 15 m)

Smart Prey:

vs. 2 Smart Predators,

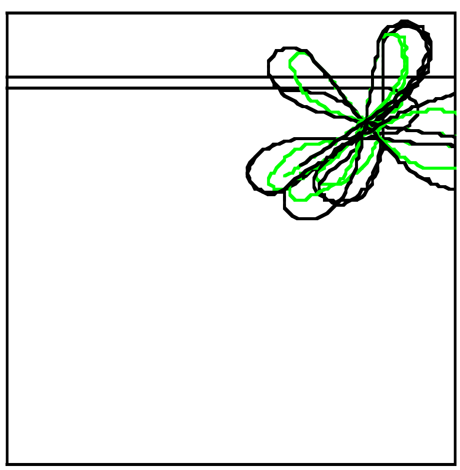


vs. 2 Hungry Predators

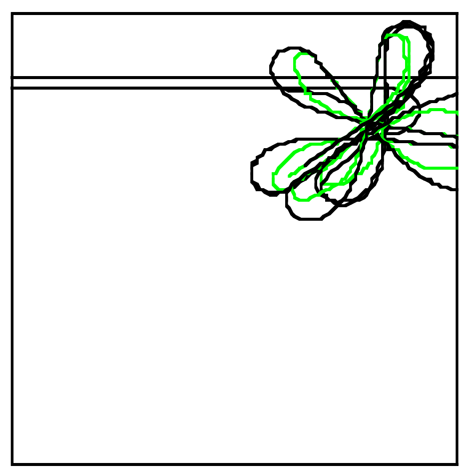


Gradient Prey:

vs. 2 Smart Predators,



vs. 2 Hungry Predators



Computational Results:

If the animals started running at a separation with the beta distribution, the overall survival rates (out of 2,000 runs each) and their standard errors were:

1-on-1:

vs.	Hungry Predator	Smart Predator
Frightened Prey	3.20 ±0.39%	3.85 ±0.43%
Smart Prey	35.7 ±1.1%	35.5 ±1.1%
Gradient Prey	35.7 ±1.1%	35.5 ±1.1%

2-on-1:

vs.	2 Hungry Predators	2 Smart Predators
Frightened Prey	3.20 ±0.39%	3.85 ±0.43%
Smart Prey	3.70 ±0.42%	4.00 ±0.44%
Gradient Prey	3.45 ±0.41%	9.40 ±0.65%

We can see from these tables how well each strategy worked:

Frightened Prey: Always fleeing the predator unconditionally leads to the lowest survival rates. This is a bad choice.

Smart Prey: The smart prey, which darts to the side when the predator closes to some critical distance behind him, did quite well in the single-predator runs. By the numbers, this is the best overall evasion strategy for the Thescelosaurus.

Gradient Prey: As discussed above, the gradient prey is the same as the smart prey for the single-predator runs. On the two-predator runs, the gradient prey did quite well against the smart predators, but more poorly than the smart prey against the hungry ones!

Hungry Predator: This predator killed the most prey in both the single-predator and hungry-predator scenarios. Because he's never mislead by the prey's movements, he relentlessly brings them down better. Our data indicates that this is the best overall hunting strategy for the Velociraptor— always head straight for where the prey currently is.

Smart Predator: This predator is too easily mislead by the prey's movements, and, except alone against smart prey, is always worse than the hungry predator. He is especially poor in the two-predator scenario against the gradient prey.

Computer Model Overview and Assumptions:

In the computer model, we decided to use physics-based motion, where accelerations were finite, instead of the more-theoretical motion of the mathematical model (in which acceleration is not considered). While there is no explicit limit on the turn radius (it isn't even calculated in the simulation), when we restrict our creatures' acceleration to the assumed value, then at top speed the animals can turn at no less than their minimum turning radius.

Hence, although the mathematical and computation models' methods differ, our results (minimum turn radius at top speed) are the same. This seemed to recur throughout the course of our numerical and mathematical experimentation— the mathematical and computational models would attack the same problem from different directions, and end up with the same answer.

Computer Model Assumptions:

1.) The predator (or predators) will slowly sneak up on the prey as long as the prey doesn't start to run away. At some distance hi (where 15m<hi<50m), the prey will notice that it is being approached and start running. This is where the simulation starts.

2.) Each animal has a maximum speed.

3.) Each animal has a maximum acceleration, derived from the equation from physics:

$$a=v^2/r$$

where v is the animal's maximum speed (see assumption 2) and r is their minimum turn radius at top speed.

4.) At each instant, each animal figures out where they would like to apply their acceleration, based on their present position, and the positions of every other animal. However, they only know the position of the other animals 20 milliseconds (their reaction time) ago.

5.) Whenever a predator stops getting closer to the prey (i.e., the predator has made a close pass by the prey) a probability-of-death function of the smallest distance between the predator and prey is evaluated (the prey then has a p(x) chance of death). That function is

$$p(x)=1/(1+4*x^4)$$

where x is the predator's closest approach to the prey, and p(x) is the prey's chance of death.

6.) Simulation continues until the prey dies, or 15 seconds elapse.

Conclusion:

Since our models are both able to produce probability of survival rates, we were able to graphically find which of the above factors most influences survival. We discovered that, despite rather large differences between our mathematical and computational models, in both models the single most important factor in determining survival rates is the distance between the prey and predator at the moment the prey detects him, h_i.

This is because the most successful prey strategy was to flee the predator directly until it closed to other critical distance h_b (1.619 m), then make a sharp turn. By utilizing its smaller turn radius in this fashion, the prey can get slightly ahead of the predator. The predator rapidly closes this distance, whereupon the prey can pull the same trick again. Each time he does so, however, he takes another chance that the predator will kill him. Hence the number of breakaway turns he has to make is the single most important factor in determining his chances for surviving one predator. His chances of survival are very low (<4%) in the two-predator case, when one predator trails the first by 8 m.

On the following pages, you will find graphs of the probability of survival versus [hi]. Figure 11 is predicted probability of survial graph produced by the mathematical model, for the case of a smart prey being pursued by a hungry hunter. Figure 12 is the graph of the actual probability, produced with 50,000 runs of the computer model for various [hi]s. The graphs, aside from the static produced by the random variation of our small sample size, are nearly identical.

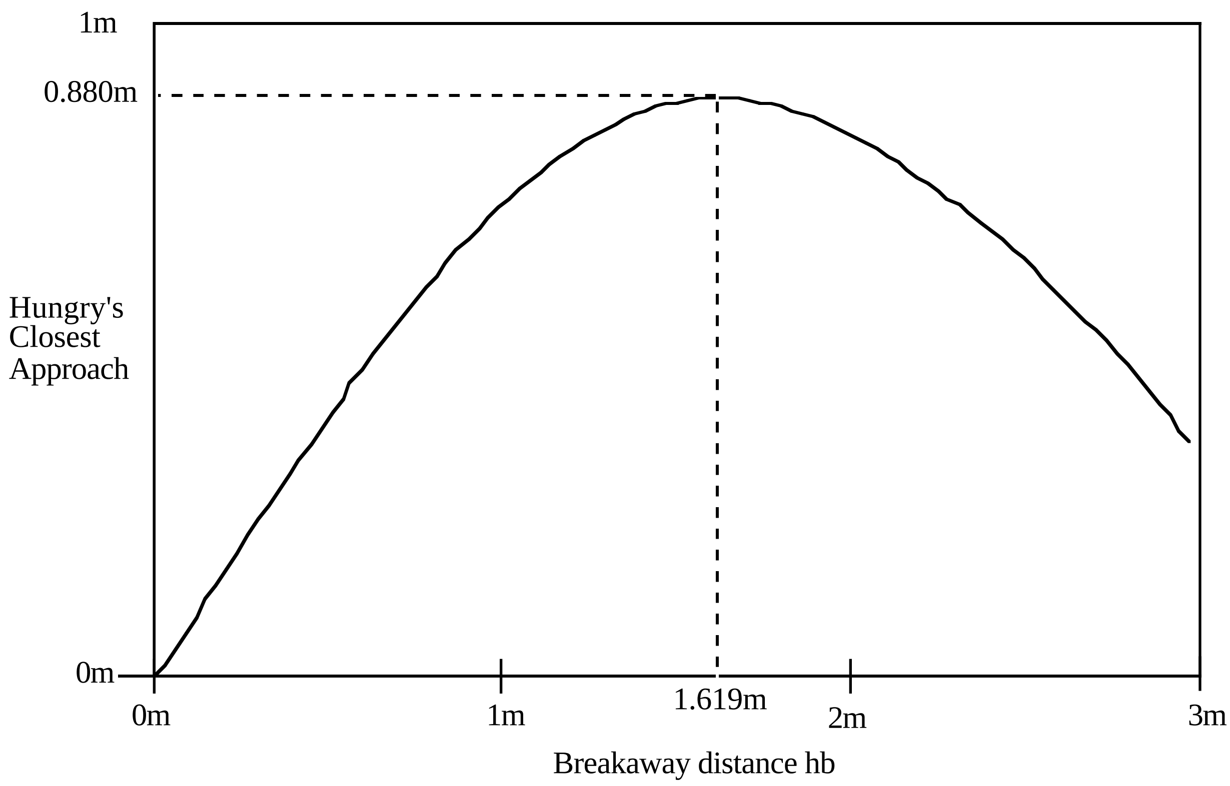
Using the mathematical model, we found that while changing the turn radii affected the sharpness of curvature of the probability graphs, it did not diminish the importance of [hi]. Using the computational model, we tried many different hunting and evasion strategies, and still [hi] was the most important factor.

Hence, it seems that in the final analysis, the prey's most important task is to be as paranoid as possible, and detect the predator as soon as possible. The predator's most important task is to blend in as well as possible, and not be detected.

Finding the breakaway distance hb:

In both the one- and two-predator models, the prey is often chased down. Since the predator(s) are faster, if the prey doesn't veer sharply from a straight line, it will die (this is just what the frightened prey does). To avoid this fate, both the smart and gradient prey veer sharply to the left or right if the nearest predator is less than some optimal distance hb away.

We found the optimal value for hb by plotting it versus the closest approach a hungry predator makes.



Hence, we can see that versus a hungry predator, if the prey makes his turn to the left or right when the predator is exactly 1.619m away, the prey will have the best probability of sucess.

A similar curve was plotted for the smart predator, and though the value for hb was nearly the same, the predator got closer.

Simulator Code in C++:

Salient Features:

Models several hunting and evasive strategies.
Models NxM predator/prey situations.
Models elevation.
Models reaction time.

Program code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "Plot.h"//Proprietary graphical output library

//Program Control:
const char *animalString="sH";//lowercase--prey.  Uppercase--predator.
//s,S=smart    f=frightened prey    H=hungry predator   t,T=turning

typedef enum {drawActivity,drawGraph}drawWhatType;
const drawWhatType drawWhat=drawActivity;

typedef enum {doActivity,doHistogram}doWhatType;
const doWhatType doWhat=doHistogram;
const int numHistoBuckets=100;
real histo[numHistoBuckets];

//Constants:
/*Critical animal parameters: speeds are in m/s, accellerations in m/(s*s),
reaction times in secs.*/
const real maxPredKMPH=60,predTurnRadius=1.5,predReactTime=0.02;
const real maxPreyKMPH=50,preyTurnRadius=0.5,preyReactTime=0.02;
const real maxTime=15;//run is for 15 seconds.
const real gravAccelleration=9.80;//in m/(s*s)
const long steps_per_sec=1000;//simulation steps per second

const real timeStep=1.0/steps_per_sec;
const real maxReactionTime=0.03;
const long maxReactionSteps=maxReactionTime*steps_per_sec;
const real maxPredSpeed=maxPredKMPH/3.6;
const real maxPredAccell=maxPredSpeed*maxPredSpeed/predTurnRadius;//a=v^2/R
const real maxPreySpeed=maxPreyKMPH/3.6;
const real maxPreyAccell=maxPreySpeed*maxPreySpeed/preyTurnRadius;//a=v^2/R
const long predReactSteps=predReactTime*steps_per_sec;
const long preyReactSteps=preyReactTime*steps_per_sec;

//Global:
int drawThisRun=0;

//Type definitions:
typedef struct point { real x,y;} point;
#define vector point

//Elevation Utilities:
real *elevation=NULL;

void readInElevationFile(char *name,real scaleBy,short drawAfterwards);
real getHeight(real inx,real iny);
vector getGravAccelleration(point p);
```

```
void readInElevationFile(char *name,real scaleBy,short drawAfterwards)
{
	if (!name)
		return;
	FILE *f=fopen(name,"r+");
	if (!f)
	{
		printf("Elevation file %s not found.  Continuing anyway.\n",name);
		return;
	}
	elevation=(real *)malloc(sizeof(real)*10*10);
	real maxElev=-10000000;
	for (int y=0;y<10;y++)
		for (int x=0;x<10;x++)
		{
			float inValue;
			fscanf(f,"%f",&inValue);
			inValue*=scaleBy;
			elevation[y*10+x]=inValue;
			if (inValue>maxElev)
				maxElev=inValue;
		}
	fclose(f);
	real maxInverse=1/maxElev;
}
real getHeight(real inx,real iny)
{//return a bilinearly interpolated elevation.
	real x=(inx+250)/500*9;
	real y=(iny+250)/500*9;
	if (x<0) x=0;
	else if (x>=9) x=8.9999999;
	if (y<0) y=0;
	else if (y>=9) y=8.9999999;
	int ix=x,iy=y;
	real fx=x-ix,fy=y-iy;
	register real e00=elevation[iy*10+ix+00],
		e10=elevation[iy*10+ix+01],
		e01=elevation[iy*10+ix+10],
		e11=elevation[iy*10+ix+11];
	return (e11*fx+e01*(1-fx))*fy+(e10*(fx)+e00*(1-fx))*(1-fy);
}
vector getGravAccelleration(point p)
{
	vector ret;
	if (elevation)
	{
//Force vector due to surface reaction=Gradient at surface*Gravitational accelleration*Mass
//(But because mass cancels out of the final motion equations, we're sticking with
//  accelleration);
		ret.x=(getHeight(p.x+0.5,p.y)-getHeight(p.x-0.5,p.y))
			*gravAccelleration;
		ret.y=(getHeight(p.x,p.y+0.5)-getHeight(p.x,p.y-0.5))
			*gravAccelleration;
	} else {ret.x=ret.y=0;}
	return ret;
}
```

```
//Vector/point utilities:
#define sqrt(x) (double)(sqrt((float)(x)))
real ptDist(const point &a,const point &b);
real ptDist(const point &a,const point &b)
{
	real xDel=a.x-b.x,yDel=a.y-b.y;
	return sqrt(xDel*xDel+yDel*yDel);
}
real vecLen(const vector &v);
real vecLen(const vector &v)
{
	return sqrt(v.x*v.x+v.y*v.y);
}
void limitVector(vector *v,const real maxLen);
void limitVector(vector *v,const real maxLen)
{
	real vecLength=vecLen(*v);
	if (vecLength>maxLen)
	{//scale the vector back so its length is maxLen.
		real scaleFactor=maxLen/vecLength;
		v->x*=scaleFactor;
		v->y*=scaleFactor;
	}
}
#undef sqrt
//Probability:
real getNormalRand(void);
inline real getNormalRand(void){return (real(rand())/real(RAND_MAX));}
real getRandProd(int numProducts);
real getRandProd(int numProducts)
{
	real ret=1;
	for (int i=0;i<numProducts;i++)
		ret*=getNormalRand();
	return ret;
}
real getRand(void);
real getRand(void) {return getNormalRand()*2-1;}
int did_I_die(real closestApproachToPredator);
int did_I_die(real x)
{
	real probabilityOfDeath=1.0/(1.0+4.0*x*x*x*x);
	if (getNormalRand()<=probabilityOfDeath)
		return 1;
	else return 0;
}
```

```
//The Animal Class:
class animal
{
      public:
            int isAlive,isPrey;
            real totalDistanceTravelled;
            point oldDrawnPos,pos;
            vector vel;
            real topSpeed,maxAccell;
            color animalColor;

            point oldPositions[maxReactionSteps];
            long oldPosIndex;
            long reactionSteps;
#define getReactPos(anim) anim->oldPositions[(anim->oldPosIndex \
                        +reactionSteps)%maxReactionSteps]
#define getReactPos2(anim) anim->oldPositions[(anim->oldPosIndex \
                        +reactionSteps+1)%maxReactionSteps]

            animal(real x,real y,real xv,real yv)
            {
                  isAlive=1;
                  pos.x=x;
                  pos.y=y;
                  oldDrawnPos=pos;
                  vel.x=xv;
                  vel.y=yv;
                  totalDistanceTravelled=0;
                  oldPosIndex=maxReactionSteps-1;
                  for (int i=0;i<maxReactionSteps;i++) oldPositions[i]=pos;
            }
            void advance(const vector &force)//go in the direction of force.
      {
                  vector accell=force;
//scale accelleration so that we're always running at full speed
                  accell.x*=10000;
                  accell.y*=10000;
//limit accelleration to the animal's maximum
                  limitVector(&accell,maxAccell);
                  vector gravAccell=getGravAccelleration(pos);
//use Euler's method to find new position
                  pos.x+=vel.x*timeStep;
                  pos.y+=vel.y*timeStep;
                  oldPosIndex--;//update reaction time index
                  if (oldPosIndex<0) oldPosIndex=maxReactionSteps-1;
                  oldPositions[oldPosIndex]=pos;
                  totalDistanceTravelled+=vecLen(vel)*timeStep;
//use Euler's method on the velocity
                  vel.x+=(gravAccell.x+accell.x)*timeStep;
                  vel.y+=(gravAccell.x+accell.y)*timeStep;
//Clip the velocity to the maximum speed
                  limitVector(&vel,topSpeed);
            }
            virtual void decide(animal **animals,int numAnimals) {}
            void draw(void)
            {
                  PlotColor(animalColor);
```

```
                PlotStart(oldDrawnPos.x,oldDrawnPos.y);
                PlotPoint(pos.x,pos.y);
                oldDrawnPos=pos;//Save old Position
            }
            virtual void die(void) //kills the simulated animal
            {
                isAlive=0;
                if (drawThisRun)//Mark a big black X where we died.
                {
                    PlotColor(black);
                    PlotStart(pos.x+10,pos.y+10);
                    PlotPoint(pos.x-10,pos.y-10);
                    PlotStart(pos.x-10,pos.y+10);
                    PlotPoint(pos.x+10,pos.y-10);
                }
            }
            virtual void destroy(void) //destroys the C++ object
            {
                delete this;
            }
};
//The Prey class: animals that can be eaten.
class prey:public animal{public:

typedef enum {predGettingCloser,predGettingFarther} predSlopeType;
        real lastPredDist;
        predSlopeType slope;
        prey(real x,real y,real xv,real yv):animal(x,y,xv,yv){
            animalColor=green;topSpeed=maxPreySpeed;maxAccell=maxPreyAccell;
            isPrey=1;reactionSteps=preyReactSteps;
            lastPredDist=100000000;slope=predGettingCloser;
        }
        virtual void flee(animal *predator,real distance)=0;
        virtual void decide(animal **animals,int numAnimals)
        {
        //Find the distance to the closest predator
            real closestPredatorDist=100000000;
            real trueClosestDist=10000000;
            animal *closestPredator=NULL;
            for (int i=0;i<numAnimals;i++)
                if (!animals[i]->isPrey)//(only worry about predators).
                {
                    real
distToPredator=ptDist(getReactPos(animals[i]),pos);
                    if (distToPredator<closestPredatorDist)
                    {
                        closestPredatorDist=distToPredator;
                        closestPredator=animals[i];
                    }
                    real trueDist=ptDist(animals[i]->pos,pos);
                    if (trueDist<trueClosestDist)
                        trueClosestDist=trueDist;
                }
        //Check to see if we've been killed.
            if (trueClosestDist<(lastPredDist))//a predator is gaining on us
                slope=predGettingCloser;
            else if (trueClosestDist>(lastPredDist))//we're getting farther
```

```
away from the predator
		{
			if (slope==predGettingCloser)
			{//we've had a minima in the last step, so we now
				//take the chance of getting killed.
				if (drawThisRun)
				{
					PlotColor(blue);
					#define boxSize 0.5
					PlotBox(pos.x-boxSize,pos.x+boxSize,
						pos.y-boxSize,pos.y+boxSize);
				}
				if (did_I_die(lastPredDist))
				{
					die();//if we lost the gamble, we're dead.
					return;
				}
			}
			slope=predGettingFarther;
		}
		lastPredDist=trueClosestDist;
	//Otherwise, we have to flee the closest predator.
		if (closestPredator)
			flee(closestPredator,closestPredatorDist);
	}
};
//scared prey flee straight away from predator
class scaredPrey:public prey {public:
	scaredPrey(real x,real y,real xv,real yv):prey(x,y,xv,yv) {}
	virtual void flee(animal *predator,real distance)
	{
		vector away;
		away.x=pos.x-getReactPos(predator).x;
		away.y=pos.y-getReactPos(predator).y;
		advance(away);
	}
};
//Smart prey flee straight away when predator
//is far away, but veer off to the side when he's close.
class smartPrey:public prey {public:
	int headedLeft;
	smartPrey(real x,real y,real xv,real yv):prey(x,y,xv,yv) {headedLeft=0;}
	virtual void flee(animal *predator,real distance)
	{
		vector accell,away,perp;
		away.x=pos.x-getReactPos(predator).x;
		away.y=pos.y-getReactPos(predator).y;
		perp.x=away.y;
		perp.y=-away.x;
		if (headedLeft)
		{
			perp.x*=-1.0;
			perp.y*=-1.0;
		}
		if (distance<1.619)
		{//Imminent death! Better veer now.
			accell.x=perp.x;//-away.x*0.5;
```

```
                        accell.y=perp.y;//-away.y*0.5;
                        advance(accell);
                } else {//A big separation-- run away!
                        headedLeft=!headedLeft;//Randomize next bolt direction.
                        advance(away);
                }
        }
};
//Turning prey run in little clockwise loops.
class turningPrey:public prey {public:
        turningPrey(real x,real y,real xv,real yv):prey(x,y,xv,yv) {}
        virtual void flee(animal *predator,real distance)
        {
                vector accell;
                accell.x=vel.y;
                accell.y=-vel.x;
                advance(accell);
        }
};

//The Predator class
class predator:public animal {public:
        predator(real x,real y,real xv,real yv):animal(x,y,xv,yv)
        {       animalColor=red;topSpeed=maxPredSpeed;maxAccell=maxPredAccell;
                isPrey=0;reactionSteps=predReactSteps;}
        virtual void chase(animal *prey,const vector &toPrey)=0;
        virtual void decide(animal **animals,int numAnimals)
        {
        //Find the closest prey...
                real closestPreyDist=100000000;
                animal *closestPrey=NULL;
                vector toClosestPrey;
                for (int i=0;i<numAnimals;i++)
                        if (animals[i]->isPrey)
                        {
                                vector toPrey;
                                toPrey.x=getReactPos(animals[i]).x-pos.x;
                                toPrey.y=getReactPos(animals[i]).y-pos.y;
                                real distToPrey=vecLen(toPrey);
                                if (distToPrey<closestPreyDist)
                                {
                                        closestPreyDist=distToPrey;
                                        toClosestPrey=toPrey;
                                        closestPrey=animals[i];
                                }
                        }
        //... then chase it (however we happen to chase it).
                if (closestPrey)
                        chase(closestPrey,toClosestPrey);
        }
};
```

```
//Hungry predators run straight for prey.
class hungryPredator:public predator {public:
	hungryPredator(real x,real y,real xv,real yv):predator(x,y,xv,yv) {}
	virtual void chase(animal *prey,const vector &toPrey)
	{
		advance(toPrey);
	}
};
//Smart predators predict the future position
//of their prey.

class smartPredator:public predator {public:
	smartPredator(real x,real y,real xv,real yv):predator(x,y,xv,yv) {}
	virtual void chase(animal *prey,const vector &toPrey)
	{
		point nowPos=getReactPos(prey);
		point lastPos=getReactPos2(prey);
		vector velPrey,predictedPrey;
		velPrey.x=nowPos.x-lastPos.x;
		velPrey.y=nowPos.y-lastPos.y;
	//Compute quadratic coefficients for prey predictor:
		const real
rSqOverTSq=maxPredSpeed*maxPredSpeed/(maxPreySpeed*maxPreySpeed);
		real a=(velPrey.x*velPrey.x+velPrey.y*velPrey.y)*(1-rSqOverTSq);
		real b=2*velPrey.x*toPrey.x+2*velPrey.y*toPrey.y;
		real c=toPrey.x*toPrey.x+toPrey.y*toPrey.y;
		real d=b*b-4*a*c;
		if ((d<0.0)||(a==0.0))
			advance(toPrey);
		else
		{
			real t=(-b+sqrt(d))/(2*a);//quadratic time of intercept
			predictedPrey.x=velPrey.x*t+toPrey.x;
			predictedPrey.y=velPrey.y*t+toPrey.y;
			advance(predictedPrey);
		}
	}
};

//Turning predators run in larger clockwise loops.
class turningPredator:public predator {public:
	turningPredator(real x,real y,real xv,real yv):predator(x,y,xv,yv) {}
	virtual void chase(animal *prey,const vector &toPrey)
	{
		vector accell;
		accell.x=vel.y;
		accell.y=-vel.x;
		advance(accell);
	}
};
```

```
//The main simulator loop: run all the animals around until either all
//the prey are dead, or our 15 seconds are up.
void runFor15s(animal **animals,int numAnimals);
void runFor15s(animal **animals,int numAnimals)
{
	for (long time=0;time<maxTime*steps_per_sec;time++)
	{
		int livingPreyExist=0;
		for (int animalNo=0;animalNo<numAnimals;animalNo++)
		{
			animal *thisAnimal=animals[animalNo];
			if (!thisAnimal->isAlive)
				continue;
			thisAnimal->decide(animals,numAnimals);
			if (thisAnimal->isPrey)
				livingPreyExist=1;
		}
		if (!livingPreyExist)//out of prey-- it's over.
			return;
		if (drawThisRun)
			if ((time&0x0f)==0)
				for (int drawNo=0;drawNo<numAnimals;drawNo++)
					if (animals[drawNo]->isAlive)
						animals[drawNo]->draw();
	}
}

animal *animals[10];
int numAnimals;
int killedPrey=0,totalPrey=0;

void createAnimals(real minSeparationBtwPredatorAndPrey);
void createAnimals(real separation)
{
	numAnimals=strlen(animalString);
	for (int i=0;animalString[i];i++)
	{
		real px=0,py=0;
		const real predSep=5;//Distance between each predator
		if (!doWhat==doHistogram)
			{px=getRand()*20;py=getRand()*20;}
		switch(animalString[i])
		{
	//s,S=smart    f=frightened prey    H=hungry predator   t,T=turning

case 's':animals[i]=new smartPrey(separation,py,0,0);totalPrey++;break;
case 'f':animals[i]=new scaredPrey(separation,py,0,0);totalPrey++;break;
case 't':animals[i]=new turningPrey(separation,py,0,0);totalPrey++;break;
case 'S':animals[i]=new smartPredator(-(i-1)*predSep,py,0,0);break;
case 'H':animals[i]=new hungryPredator(-(i-1)*predSep,py,0,0);break;
case 'T':animals[i]=new turningPredator(-(i-1)*predSep,py,0,0);break;
			default:
				printf("Unrecognized animal type: %c.\n");
				exit(0);
		}
	}
}
```

```
void destroyAnimals(void);
void destroyAnimals(void)
{
	for (int i=0;i<numAnimals;i++)
	{
		if (animals[i]->isPrey&&!animals[i]->isAlive)
			killedPrey++;
		animals[i]->destroy();
	}
}
//getBetaH and getGammaH return an inital separation,
//	appropriately distributed.
real getBetaH(void);
real getBetaH(void)
{
	const int v=3,w=4;
	real m=-log(getRandProd(v)),n=-log(getRandProd(w));
	return 15.0+35.0*m/(m+n);
}
real getGammaH(void);
real getGammaH(void)
{
	const int a=6,b=5;
	return -b*log(getRandProd(a));
}
//Main runs and plots some number of simulation runs, possibly plotting them.
main()
{
	srand(2);
	readInElevationFile(NULL,0,0);//"Mountain.elev",2000,0);
	const real maxCoord=150;
	PlotAxes(-maxCoord,maxCoord,-maxCoord,maxCoord,300,300);
	for (long whichRun=0;whichRun<500;whichRun++)
	{
		createAnimals(getGammaH());
		if (drawWhat==drawActivity)
			drawThisRun=whichRun%20==0;
		runFor15s(animals,numAnimals);
		destroyAnimals();
	}
	if (drawWhat==drawActivity)
		printf("Survival Rate: %.3f%%\n",(1-
real(killedPrey)/real(totalPrey))*100.0);
	return 0;
}
```

Appendix D:
Rather uninteresting results:

Our computer model was capable of handling any reaction time, and any elevation map. However, preliminary experimentation with these factors did not lead to any interesting or useful results.

Making the reaction time much longer (greater than 0.1 s) tended to hurt the predator's chances of sucess (especially the smart predator, which is then predicting based on older data), but this can probably be attributed to mere increased difficulty in locating anything when you're moving at 60 kmph and you can only see where things were 0.1 s (or 1.6 meters) ago. So this factor didn't seem very crucial or interesting.

Elevation, while having many interesting possiblities for research, is usually too dependant on specific terrain features combined with pre-hunt predator collusion (e.g. to run the prey into a narrowing, steep-walled valley) to produce any generally useful results. The only general observations we can offer is that if there is a steep peak between the predator and prey, it makes the predator's job harder (and decreases his chances of a kill); and if the predator gets to start down the slope of a bowl, and the prey has to climb up the slope, that makes the prey's job harder (and increases his chances of a kill).

Elevation would be a very interesting aspect of a site-specifice modelling problem, but tended not to lead to any general results.