1-1-1991

BUGS: A Test Bed For Catastrophe Based Collective Behavior

Thomas L. Clarke

INSTITUTE FOR SIMULATION AND TRAINING

May 1991

BUGS: A Test Bed for Catastrophe Based Collective Behavior

Thomas L. Clarke

Institute for Simulation and Training
12424 Research Parkway, Suite 300
Orlando FL 23826

University of Central Florida
Division of Sponsored Research

IST-TR-91-17

INSTITUTE FOR SIMULATION AND TRAINING

BUGS: A Test Bed for Catastrophe Based Collective Behavior

Contract N61339-89-C-0044
May 1991

IST-TR-91-17

Prepared by

Thomas L. Clarke

Reviewed by

Margaret Loper

Institute for Simulation and Training • 12424 Research Parkway, Suite 300 • Orlando, Florida • 32826

University of Central Florida • Institute for Simulation and Training

Table of Contents

INTRODUCTION

IST researchers have constructed a model environment to rapidly generate and test ideas about force-field driven, catastrophe switched behavior . While the ultimate goal of this project is to help produce a dismounted infantry semi-automated force (SAFOR) for the simulator networking (SIMNET) environment, the SIMNET environment is inherently very complex. The simulated environment facilitates the testing of ideas for behavioral modeling.

The program, Bugs, was used to test the ideas reported on in the paper "Human Behavioral Modeling Using Catastrophe Theory" presented at the 2nd IST Behavioral Representation Symposium. (Appendix 1 contains this paper.)

Bugs was implemented on a NeXT Workstation equipped with a Motorola 68040 processor. The language used was Absoft Objective Fortran. Fortran was chosen because of its ability to include complex numbers and its good match to numeric processing. The Absoft version was used because it included objective extensions that allowed easy implementation of graphics routines. See Appendix 1 for more detailed information.

The work described in this report was partially funded through DARPA/PM TRADE contract N61339-89-C0044, Intelligent Simulated Forces Evaluation and Exploration of Computational and Hardware Strategies.

DISCUSSION

The following discussion explains the contents of the source code files needed to compile a working version of the Bugs program for the NeXT workstation. This code works with version 2.0 of the NeXTStep operating system. It is not known whether it will work with version 1.0. Listings of these files are included as Appendices.

Makefile

UNIX style file controlling program compilation and linkage. The variable *FFILES* contains the names of dependent code files.

Bugs.inc

Block of code that declares variable in common to all the Bugs programs. All the variables – except the parameter variables *nx, nxm, ncom*b – are commoned in the block GLOBALS. This use of common forces within objective Fortran makes these variables static.

The variable names are meant to be suggestive of their function, but a brief explanation will be helpful.

nx (nxm)	- size of playing grid (minus 1, need for array sizing)
meanx,meany	- fossils, defined but not used
npoten	- controls display of potential function
ngen	- number of generations or time steps run
nodisp	- toggle variable to turn off display
liveb(livef)	- number of buddy (foe) agents left alive

stopped - toggle variable to pause simlation
ncomb - maximum number of agents (array dimension)
xmax,ymax,x0,y0 - maximum and minimum coordinates
dx,dy - grid spacing in screen units
emax - floating point *nxm*
budr,budi (foer,foei) - real and imaginary parts of buddy (foe) potentials
wbudr,wbudi (wfoer,wfoei) - real and imaginary parts of buddy (foe) weights

Imaginary parts are used to calculate cross-potentials, e.g. foe to bud

xb,yb (xf,yf) - x and y coordinates of buddy (foe) agents
stateb(statef) - state of buddy (foe) agents
budbud(budfoe) - pairwise potential between buddy and buddy (foe)
foebud,foefoe - pairwise potential between foe and foe(buddy)

These are not now used but would be needed in SIMNET.

Bugs.f

Main program - sets up NeXT window objects and main timing loop. This program is a modification of an example provided with the Absoft compiler and some of the peculiarities of syntax reflex this genesis.

For those unfamiliar with objective Fortran syntax, the statements beginning with @ are objective syntax and follow closely the analogous statement in objective C.

The

@INTERFACE Bugs : View

line specifies the messages to which the object *Bugs* (which is of class *View)* responds. Bugs responds to messages *@- suspend, @- gofast, @- step, @- density, @- drawSelf* as specified by the subsequent lines of code. The *@- mouseDown* message was not implemented. It would have been desirable to allow mouse driven editing of the agents within the Bugs View, but time precluded implementing this method.

The *@IMPLEMENTATION* statement begins a section which specifies how Bugs responds to each method. The *@+ newView* section contains intialization code which places the window on the screen (*NXEraseRect*) specifies the max range of x and y, calls the *initialize* subroutine (see Bugs_subs.f) and calls the *display* subroutine (see Bugs_disp).

The strange bracketed code indicates a message being sent. For example, *[self newFrame:&r]* means *self* (this object) is being recursively commanded to respond to a newFrame message. Similarly, the *[Timer newTImer: ...]* message implements the basis timing loop of the program by sending the *Timer* a message (see Timer.f). The commented message ... setFlip ... is left over from the Absoft prototype. It causes the coordinate system to be flipped vertically which is undesirable when using some fonts on the screen.

The sections beginning *@- step, ..., @- drawSelf* contain the code that implements the various methods. Most of these methods just change the state of a switch variable in response to a mouse click on a menu item. The method *drawSelf* does the work. The subroutine *neighbor* is called to calculate the potential function; then rule is called twice to update the states of the agents: once for friends and once for foes.

The code beginning with *PROGRAM* main is standard NeXT code for starting a program. A new application *NXApp* is established by messaging *Application*; *setUp* code is called; the application is given a *run* message and then a *free* message after completion.

The subroutine *setUp* sets up a needed *rect* window data structure and creates a window view with a message to *Window*. *myWindow* is modified and enhanced by sending it various messages. The *myPanel* window for the menu items is set up by a message to *Panel*, and is subsequently modified. The final section defines the actions in response to various mouse clicks by sending *myMenu* various *addItem* messages.

Briefly these actions are:

Info	- the usual NeXT program information message
Pause	- stop program execution but retain display, second click toggles
GoFast	- continue simulation but turn off display for speed, toggles
Single Step	- execute one time step of the simulation
Show Density	- display friend-friend potential function as shades of gray, subsequent clicks cycle through friend-foe, foe-foe, foe-friend and off
Print	- invokes the standard NeXT printpanel
Hide	- miniaturizes the display window, program continues to run
Quit	- stop program

Bugs_disp.f

This file contains two display subroutines. Both display routines make use of calls to NeXT display Postscript screen commands. These calls are identified by the prefix PS and have the effects suggest by their names. One peculiarity is the the use of the VAL() function which is needed since the native NeXT C-PS functions use call by value whereas Fortran uses addresses. The other thing to note is that the basic unit of Postscript drawing is the path, hence the calls to *PSnewpath* and *PSclosepath*.

The first subroutine *Box* draws a nice box around the playing field of the size specified by its arguments.

The *display* subroutine displays the generation number, the number of agents left alive, and optionally, the positions of agents and the potential functions. Control is basically by nested if statements.

The non-Fortran programmer will be mystified by the sections that first "*do i=1,100;living(i)=0;end do*" then *encode* according to some *format* and then call a *PS*-fucntion. Fortran is very bad with strings. What is happening is that the array *living* is having string data entered and then the *PS*-function is being called with that string data.

An English version of the program flow is as follows:

A call to *Box* clears the display.
If display is not off or hasn't been displayed for ten generations then display.
Display consists of:
Two calls to *Box* to make a nice nested appearance.
If potential display is on, then
loop to find max and min of desired potential
loop to display potential scaled to [0,1]
Note - display is via Postscript line drawing function with line width set to grid size, dx, so that the Postscript machinery automatically provides the gray scale display
An encode sequence sets a font, another encode then results in display of

the type of potential function.

After setting *liveb* and *livef* counters to zero, two loops display the locations of the agents using *PSarc* circle drawing commands. State of agent is indicated by shade of circle.

An encode sequence sets the font and three more encode sequences display the numbers of each type of agents remaining and the generation number. Obviously in go-fast mode the number of agents is updated only every 10th generation, although this could be modified.

Bugs_subs.f

This file includes three subroutines that implement the computations underlying the Bugs simulation: neighbor, initialize, and rule. The first, *neighbor,* calculates the pairwise potentials using an FFT calculation of the convolution. The second, *initialize,* sets up the simulation, and the third, *rule,* updates the states of the agents. The discussion begins with *initialize.*

The first section of *initialize* deals with setting up the weight arrays used in calculating the convolution from which the potential is derived. Three Gaussian distributions with standard deviation *sig{1,2,3}* are used. The Gaussian normalization factors *an{1,2,3}* are calculated once and for all, as are the exponential argument weights *as{1,2,3}*. The i,j loop then loads the arrays. As noted above, the real parts of the array are used to hold the like weights (bud-bud, foe-foe), and the imaginary parts the unlike weights (foe-bud, and bud-foe); this saves a whole set of FFTs. *FFT2DREV* is used to transform the weights to the Fourier domain once and for all.

Finally, a modular pseudo-random function is used to populate the playing grid with agents (Absoft FORTRAN had no convenient random-number generator). The debris of commented statements shows some of the history of experimentation. All agents start in the timid state (0.5).

The subroutine *neighbor* calculates potential functions using multiplication in the Fourier domain. The potential arrays are first zeroed. They are then incremented by the indicator value 1 wherever an agent is located. The transform *FFT2DREV* is then used to pass to the Fourier domain. The transformed locator function is then multiplied by the conjugate of the transformed weights to get the transformed potential function. The potential is brought back to spatial domain via *REVFFT2D.*

Subroutine *rule* begins by setting the parameters, *swfact,* and *dead* for a two dimensional linear-hysteresis approximation to the cusp catastrophe. As mentioned in the paper, a linear approximation was used rather than a cubic cusp switch surface. This was based on the intuition that what matters is the topological pattern of hysteresis not the exact details of the hysteresis.

The like-like (*foe0*) and like-unlike (*bud0*) potentials are looked up. If *like-unlike> dead * like-like* the agent is killed. If not killed, the state of the agent is calculated. In the linear-hysteresis approximation to the cusp, if *swfact<bud0/foe0<1/swfact,* the state is unchanged. If in a overwhelmingly unlike region, *bud0<foe0*swfact,* the agent becomes timid (0.5). If in a largely like region,
*foe0>bud0*swfact,* the agent become bold (2.0).

For a true cusp the code would be little changed. In this case swfact becomes a function of the distance from the origin, so that inserting the statements

sum=bud0+foe0:diff=bud0-foe0:hyst=swfact(bud0+foe0)**3,*

and modifying the ifs to read

if (diff.lt.(-hyst)) statef(i)=0.5
if (diff.gt.hyst) statef(i)=2.0

would implement a true cusp; the value of *swfact* will have to be changed correspondingly. The validity of these expressions follows from shifting to sum and difference coordinates in the (bud,foe) plane. This modification has not been tested, but intuitively the behavior should be similar to the linear version and the amount of extra computation required is minimal.

Finally the motion increments *v1* and *v2* are set according to the agent's state. The potential gradients are calculated and the agent moves according to *v1* and *v2* and the sign of the gradients.

2D_FFT.f

This file contains two dimension FFT routines. As with one dimensional routines, these routines are provided in pairs so that overhead of bit reversal and transposition is avoided. The extra switch variable *isw* is included to control this function.

For two dimensional, Fourier transforms, one dimensional transforms are first performed along one axis, then one dimensional transforms are performed along the other axis. Since Fortran (most compilers) store multi-dimensional array data first-index fastest, the address of the first element of each column can be passed to a one dimensional transform to avoid the overhead of two dimensional index calculation. If the array is then transposed, the same technique calculates the other transform direction without the overhead.

When the transforms are used for convolution, the arrays do not need to be retransposed, nor so the final transforms need to be bitreversed. This saves additional overhead. The paired routines in this file FFT2DREV and FFT2DREV can be used in this fashion. Note that the bit reversal of the first set of transforms cannot be avoided when using standard one dimensional transforms, since skipping it would lead to a very strange data order on input to the second set of transforms.

Sub-FFT.f

This file contains one dimensional FFT routines. The code is rather unremarkable textbook code except that bit reversal is not performed. Two versions are provided *FFTREV* gives bit reversed output, whereas *REVFFT* takes bit reversed input. The argument *is* determines the direction of the transform *is=+1* for reverse, *is=-1* for forward (blame the sine function for reversal).

These paired routines are useful for convolution. Convolution only requires multiplication of corresponding values in the frequency domain, so that bit reversal is not needed provided appropriately paired transforms are used. This little trick saves the overhead of bit-reversing the time series.

The routines would have been more elegant if Fortran's complex number data type had been used. Ultimate translation to C would have been more difficult, however.

bitrev.f

This textbook routine reorders time series bit reversed by FFT algorithms. Also in this file is a little integer log_2 function, *ilog*2.

Timer.f

Supplied by Absoft to implement repetitive programs.

RECOMMENDATIONS

The ultimate goal of this project is to apply the ideas reported on in the attached paper, "Human Behavioral Modeling Using Catastrophe Theory", to the dismounted infantry SAFOR applications in the SIMNET environment. The following features must be incorporated in any SIMNET SAFOR code derived from Bugs.

(1) Display - Display of agent motion is current via the routines in Bugs_disp.f. This would be replaced by calls to appropriate SIMNET routines (C-functions, whatever) that generate the necessary protocol data units (PDUs) to make agents appear at appropriate locations within the SIMNET world.

(2) Force Fields - The force fields play two roles in Bugs. They determine the direction of agent motion by via a gradient ascent algorithm, and they determine agent behavior via the catastrophe-based behavior model. In a SIMNET SAFOR, the force field concept could be used as in Bugs with the addition of "infinite" potentials to implement static features like terrain obstacles (see the paper by Hung Le at the IST BR Symposium). The computational requirements would be relatively high, however, even with FFT techniques, for higher resolution grids than the 32 by 32 used in Bugs.

Slow updates may not be a problem, since a relatively sluggish force-field response (seconds) may be acceptable in SIMNET — agent behavior would of course occur at a faster time scale than the force-field update. Imagine neighbor running slower and asynchronously from rule. To implement this neighbor would have to be modified to takes its inputs from appropriate SIMNET routines.

Perhaps a better approach would be to use the spirit rather than the letter of Bugs. The primary purpose of the force field is to provide a simple test environment for behavioral ideas. These ideas could be implemented directly into code that obtains environmental information from SIMNET routines, evaluates the agent's status using catastrophe theoretic ideas, and then displays the agent's behavior using other SIMNET routines. In this approach, all that remains of Bugs would be modified versions of initialize and rule. The modified rule would determine the density of enemies versus friends directly from calls to SIMNET routines and update state accordingly. Information from SIMNET routines would be used to determine behavior based on the state, and the behavior displayed by invoking other SIMNET routines.

Appendix 1

Human Behavioral Modeling Using Catastrophe Theory

Human Behavioral Modeling Using Catastrophe Theory

by
Thomas L. Clarke
J. Martin Otte
Institute for Simulation and Training
University of Central Florida

ABSTRACT

A simple model of human behavior applicable to force simulation and based on catastrophe theory is developed. The large number of agents that must be modeled for a dismounted infantry automated force mandates that the algorithms used be as simple as possible. Simple models in which behavior is determined by local variables are investigated. Models based on cellular automata are found not to be useful. Physically motivated models based on catastrophe theory are promising, however. This is not surprising since the seven fundamental catastrophic transitions have been used successfully to model behavior in a variety of circumstances.

For dismounted infantry, the cusp catastrophe is used to model the transition from bold behavior when surrounded by friends, to timid behavior when surrounded by enemies. In a simulated battlefield environment, infantry agents are modeled as point particles moving under rules determined by the bold or timid state. The infantry in this simple battlefield exhibit interesting, non-trivial behavior. It is thus likely that catastrophe-based behavior will be useful as the basis for more complex simulations.

INTRODUCTION

Current approaches to modeling battlefield behavior are often based on conventional symbolic Artificial Intelligence (AI). If this symbolic AI approach were to be used in our research, then a rule-driven behavior model would be implemented for each of the many agents needed for a dismounted infantry, semi-automated force. However, this symbolic AI approach requires an unmanageable and impractical amount of computing power. As an alternative then, we have chosen to consider non-symbolic models.

Perhaps the simplest such non-symbolic models are based on cellular automata such as John H. Conway's game of Life (Poundstone, 1985). In a cellular automata, space is divided into discrete squares, each of which is characterized by a state. For behavioral applications these states would represent the absence or presence, and the condition of a simulated agent. In Conway's Life, each cell is either empty (0) or full (1). Time progresses forward in discrete steps, and the state of a cell at the next time step is determined by its present state and the state of its eight neighbors. Conway's rules are simple: (1) if a cell is empty now and precisely three neighbors are full, then

the cell becomes full in the next time step; (2) if a cell is full and two or three neighbors are full, then the cell remains full in the next time step; (3) otherwise the cell becomes empty. These rules balance the problems of runaway population growth with the problems of extinction; indeed, Poundstone (1985) shows that Conway's Life automata is sufficiently rich to emulate a universal Turing machine.

Our first attempt at designing a behavior model was based on Conway's Game of Life. Life — as it is called for brevity — contains within it the concept of birth and death of agents in a large matrix of cells. The birth aspect can be thought of as an analogy to the increase in strength of an agent receiving reinforcements and becoming more aggressive. The death aspect is analagous to an agent being overwhelmed by an enemy and becoming timid and retreating.

Life, however, had some serious shortcomings in its unmodified condition. For example, there were only two cell states on the playing board: the 'on' states, which could be considered as an agent; and the 'off' states, which were simply the background environment. To introduce enemy agents, another cell state was added to the game. This cell state is conveniently represented as a third color or else a shade of gray.

There was still another drawback to unmodified Life: the rules of the game only considered the 'on' and 'off' state of cells that were directly adjacent to one another. This is a drawback because on the battlefield, the presence of all units at various distances from your own must be considered important. Life doesn't model this situation. To remedy the problem, a two dimensional Gaussian distribution of weights was introduced into the sum of neighbor states used to determine the next cell state. The neighbor sum

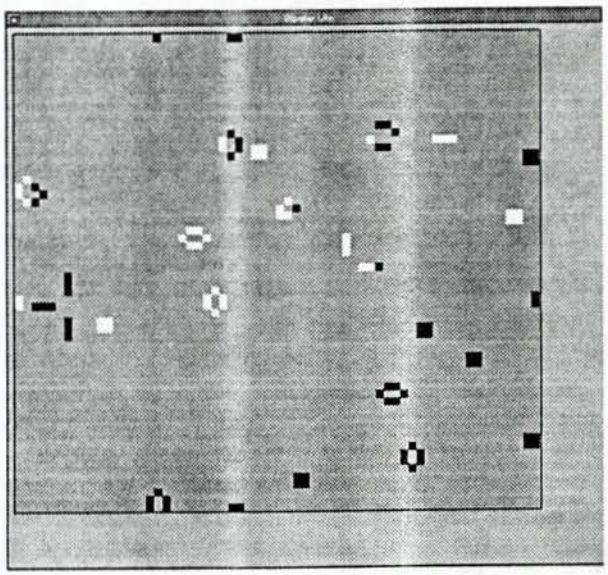


Figure *1*. *Generalization of Conway's Life to two types of agents, white and black.*

thus includes a contribution from non-adjacent cells which models the importance of the presence of units at various distances and remote positions .

Figure 1 shows a snapshot of the results of running a generalization of Conway's Life in which each cell is in one of three possible states: +1 (friendly, white), 0 (empty, gray), or -1 (enemy, black). The state of a cell is determined by the sum of the states of its neighbors. For white, the rules are Conway's; for black, the rules are Conway's with the sign reversed. There are of course many more possibilities for generalizing the rules, and several were tried. Unfortunately, the behaviors of these two-phase automata did not seem to capture the richness of behavior needed on the battlefield. As Figure 1 suggests, the results are generally a useless debris of blinkers (3 cells in a row that alternate directions from one time step to the next), blocks (stable 2 by 2 cell arrays) and other elements from Conway's bestiary. Somewhat surprisingly, the 2 by 2 block is impregnable to enemy attack. That is, a 2 by 2

black block cannot be destroyed by any combination of white cells coming in contact with it. This leads to the interesting but unrealistic prospect that a picket fence of blocks would form an impenetrable and unassailable barrier.

Additional generalizations, such as introducing the long-range Gaussian interactions, were tried but did not result in more realistic behavior. Finding a good balance between birth, death, and conflict has been difficult. The long range interactions resulted in simulations that either quickly died-out or else quickly and exponentially overpopulated.

PARTICLE-LIKE MODELS

The failure of cellular automata patterned after Life led to the consideration of other simple models for behavior. In particular, an approach based on physics-like particles, having a few simple internal states, seemed promising. This type of model retains the idea of long-range interactions with locally determined behavior, but now the paradigm is particles moving in a potential field of force.

Figures 2 and 3 show the potential fields used in the current work. The potential in Figure 2 is used to model the interaction between agents of like sign — that is, between friend and friend, and between foe and foe. An agent subject to such a potential moves so as to maximize or minimize the potential — that is, the agent moves up or down the gradient. This potential, with its central well, was chosen to provide a repulsive force at short distances: we don't want the agents to clump into small-sized units.

Figure 3 shows the potential between unlike agents (friend and foe). Because we want agents to close in for the kill when appropriate, this potential has been designed without a central dip.

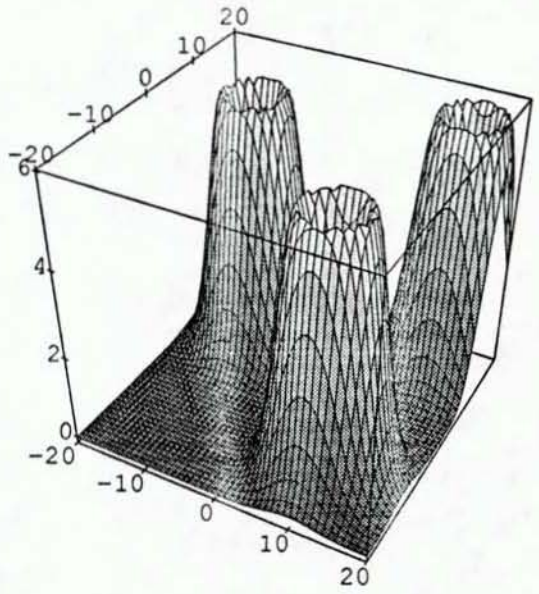


Figure 2. Potential between like agents (e.g. friend to friend).

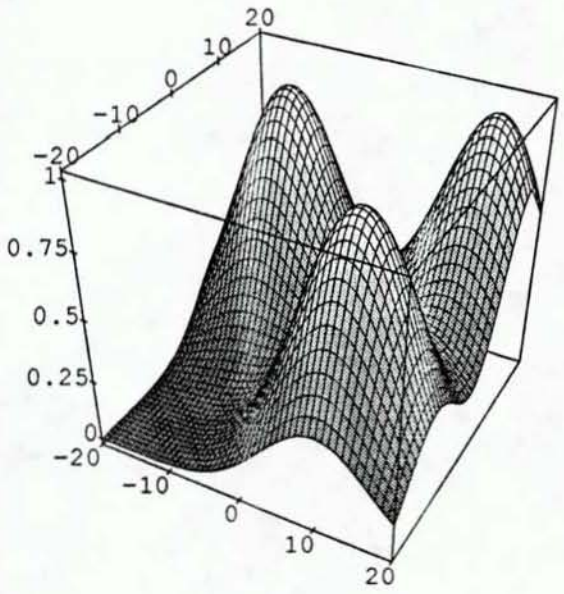


Figure *3. Potential between unlike agents (e.g. friend to foe).*

More formally, the friendly potential Φ of an agent at a location (x, y) is the sum of the separate potentials for each friendly agent:

$$\Phi(x, y) = \Sigma_i \phi(x_i - x, y_i - y); \quad (1)$$

similarly, the enemy potential Ψ is

$$\Psi(x, y) = \Sigma_i \psi(x'_i - x, y'_i - y). \quad (2)$$

Limiting the motion of agents to a discrete grid: $x=i\Delta x$, $y=j\Delta y$, $i=1,...N$, $j=1,...M$, the sums take the form of convolutions:

$$\Phi_{ij} = \Sigma_{k,l} \phi_{i+k,j+k} f_{ij}, \quad (3)$$
$$\Psi_{ij} = \Sigma_{k,l} \Psi_{i+k,j+k} e_{ij}, \quad (4)$$

Here, $\phi_{ij} = \phi(i\Delta x, j\Delta y)$, and $\psi_{ij} = \psi(i\Delta x, j\Delta y)$ are the discrete potentials; the occupancy function f_{ij} (e_{ij}) is also a descrete function if a friendly (enemy) agent is located at $(i\Delta x, j\Delta y)$, or zero.

The convolution suggests an efficient way to compute Φ and Ψ via the fast Fourier Transform (FFT) algorithm. Fourier transforming ϕ, ψ, e, and f, the convolutions become multiplications (Rabiner and Gold, 1975) :

$$\Phi = T^{-1} [T(\phi) T(f)], \quad (5)$$
$$\Psi = T^{-1} [T(\psi) T(e)], \quad (6)$$

where T and T^{-1} denote the forward and inverse Fourier transforms respectively.

Using the FFT to evaluate T on an NxM grid reduces the calculation time from order N^2M^2, to order $NM(\log_2 N)(\log_2 M)$. For $N=M=1024$, this is a saving of a factor of more than 10,000 in time.

Note that this speedup is achieved for an arbitrary number of agents because the computation time is independent of the number of agents. Of course, if the number of agents is less than $(\log_2 N)(\log_2 M)$, then evaluating the sums in equations (1) and (2) directly will likely be faster. Even for a relatively large 1024 by 1024 grid, the advantage is lost if the number of agents exceeds the modest level of 100.

CATASTROPHE THEORY

Catastrophe theory is a branch of topology that is concerned with classifying the forms that singularities of functions may take. A remarkable result proved by Rene Thom (Gilmore, 1981) shows that these singularities can only take on a limited number of forms. More precisely, an arbitrary singularity can be transformed via change of variables so that it is closely approximated by one of several basic models; no matter how weird a given catastrophe may seem, it turns out to be close to one of the basis forms. For singularities described by four or fewer parameters, there are seven basic catastrophe models. The canonical equations of these models are exhibited in Table I.

Table I
Singularity Equations for Basic Catastrophe Models

Fold:

$$1/3\, x^3 - a x$$

Cusp:

$$1/4\, x^4 - a x - 1/2\, b x^2$$

Swallowtail :

$$1/5\, x^5 - a x - 1/2\, b x^2 - 1/3\, c x^3$$

Butterfly:

$$1/6\, x^6 - a x - 1/2\, b x^2 - 1/3\, c x^3 - 1/4\, d x^4$$

Hyperbolic:

$$x^3 + y^3 + a x + b y + c x y$$

Elliptic:

$$x^3 - x y^2 + a x + b y + c (x^2 + y^2)$$

Parabolic:

$$x^2 y + y^4 + a x + b y + c x^2 + d y^2$$

In the table, a, b, c, d are the parameters, and x, y are the variables which exhibit the singularity. The thing to note is that the equations of these canonical models are simple polynomials.

Some of simpler ones have been successfully used to describe human and animal behavior in various situations (Zeeman, 1976). Of the seven catastrophes, one of the simplest to describe and understand is the cusp catastrophe. It is a geometrical object in the usual three dimension Euclidean space and looks like a folded sheet or blanket. The other catastrophes also have intuituvely appealing geometric interpretations as their names suggest.

This cusp catastrophe, which we are using to model a dismounted infantry agent's response to various battlefield situations, is shown in Figure 4. The vertical axis, x, represents the agent's behavior on an agressive to timid scale. The control parameters, a and b, are the density of friendly agents and the density of enemy agents respectively. When the agent is surrounded by friendly forces, it exhibits bold and aggressive behavior; alternatively, when the agent is surrounded by enemy agents, it shows timid and retreating behavior.

So far this behavior model may seem fairly trivial, but where catastrophe theory shines is in the transition region between the two types of behavior. The arrows in Figure 4 illustrate two possible paths of transition between aggression and timidity. The upper arrow shows how an agent may pass smoothly between aggression and timidity provided the stimuli (friendly and enemy densities) are not too strong. A good commander probably takes his troops into battle along such a path so that their responses remain predictable.

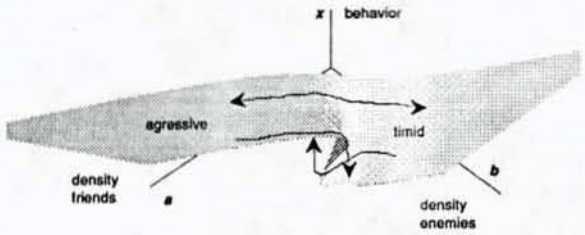


Figure *4. Cusp catastrophe as applied to modeling agent behavior.*

The lower arrows lie in regions where the stimuli are stronger and as a result cross the fold region of the cusp catastrophe. The agent's behavior now becomes discontinuous. Starting out aggressively, surrounded by friends, the agent keeps up his courage in the face of increasing enemy opposition until suddenly his nerve breaks and he turns timid. Conversely, a timid agent retreating from enemy territory remains timid until he suddenly is heartened by the presence of many friends. The catastrophe model thus introduces a degree of hysteresis into the transition between aggression and timidity.

Aggressive and timid behavior have been considered by some researchers to be influenced by two conflicting emotions: fear and rage. In the dismounted infantry simulation, these emotions are modeled by the response of the agents to various threat conditions posed by enemy troops in the

battlefield. Sometimes the battlefield conditions promote the two emotions of fear and rage simultaneously and to approximately the same degree of intensity. When this happens, the behavior of a military unit will depend on its behavior during the immediately previous time period. If, during this time period, the troops have been aggressive and the battle turns against them, they will still tend to be aggressive for a while. Likewise, if, during this time period, the troops have been retreating in fear, and they start to receive significant reinforcements, some time will be needed before they become courageous enough to attack.

The particle agents exhibit two different potential climbing modes depending on their bold/timid state. When bold, particles are attracted to friends, but are more strongly attracted to enemies; they are on the attack, but try to maintain contact with their unit. When timid, particles are attracted to friends, but are repelled by enemies; they are thus attempting to regroup while avoiding contact with the enemy. Many other variations of behavior are possible, but as shown below these produce interesting results.

RESULTS

A particle model of agents using a cusp catastrophe switch between behaviors as outline above was coded on a NeXT 25 MHz 68040-based workstation in Absoft Objective FORTRAN (yes, FORTRAN; it is still the only language with a native complex number data type). The grid used was 32 by 32 (***N=M=32***) so that several cycles of time were calculated per second and experimentation could go rapidly. The cusp catastrophe in Figure 4 was actually approximated

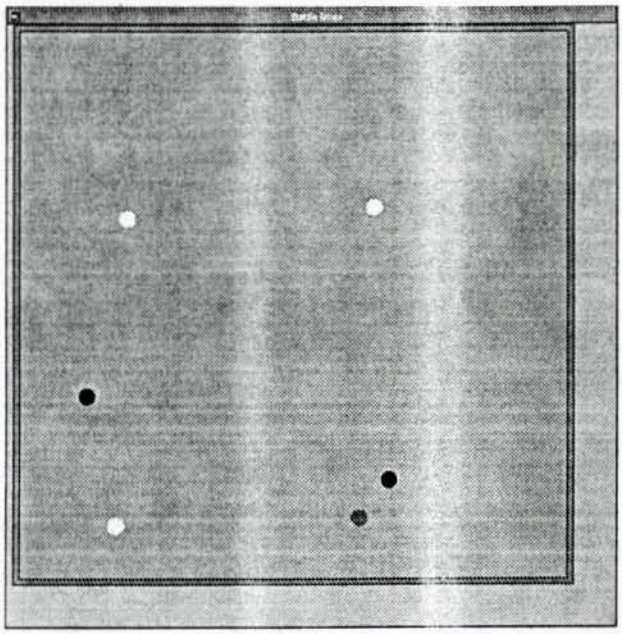
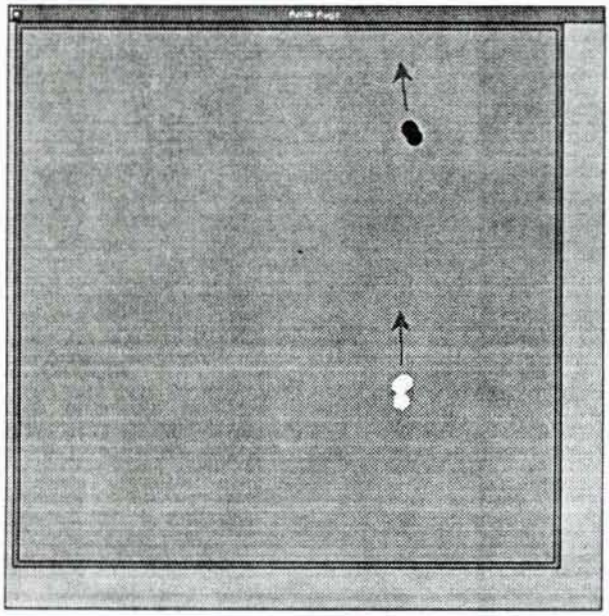


Figure 5. Forces of 3 white and 3 black agents are initially disorganized and timid.

by two overlapping planes and an additional third behavior was added. When an agent found itself in a thin strip along the enemy axis, it died and was removed from the simulation.The simulation run shown in Figures 5-8 involving three black and three white agents plays out an interesting scenario. The agents were placed at random on the grid in a timid state. Timidity is indicated in the displays by an off-color (off-white or off-black). Aggression is shown when the agent is fully black or white. Figure 5 is several time steps into the simulation and the agents are moving around trying to make contact with friends and to generally get organized. Note that for simplicity in dealing with boundaries, the simulation grid has the topology of a torus. It has no boundaries; the top of the grid is connected to the bottom and the left side is connected to the right side.



1414

Figure 6. The agents soon organize into tight clusters and become aggressive. The upward chase is on.

After several more time steps, the agents have formed tight little clusters and begin an upward trending chase. These clusters persist for two circuits of the grid.

Perhaps tiring of the chase, one black agent reverses field and, crossing the bottom "border", attacks the white group from above. The agent has no such motive, of course; its motion is merely the result of the randomly chosen initial conditions. Still, the nice anthropomorphic interpretation that can be given to its behavior suggests the possibilities of behavior modeling based on catastrophe theory.

The attacking black agent then succeeds in luring a white agent away from its group. The white agent then finds itself in the killing strip of the enemy/friendly plane where it dies and is removed from play. Already in Figure 8, this white agent has turned timid, but it is too late! The

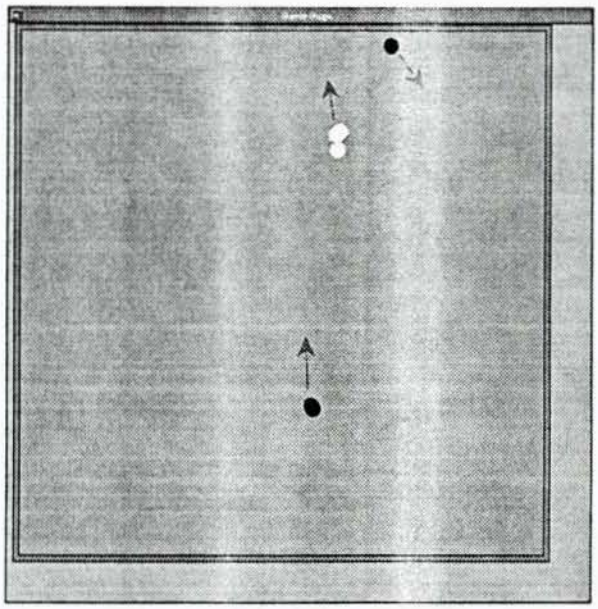


Figure 7. Black tries a flanking maneuver, one agent attacking from above.

white agent cannot get back to its group before it is killed.

This simulation ultimately results in a win by black, albeit black does lose one entity. Again, while the simulation presented here is based on very simple local dynamics, it succeeds in capturing the essence of battle.

CONCLUSIONS

The behavior model presented here shows much promise as the basis for modeling the large number of agents that would be required in a simulated dismounted infantry force. Despite the simple, local basis of the behavior, it captures many features of battle.

The computational resources needed to calculate the potential functions used to guide the particle-like entities are fairly large, but the use of fast Fourier transform techniques enables a single potential computation to serve for all agents.

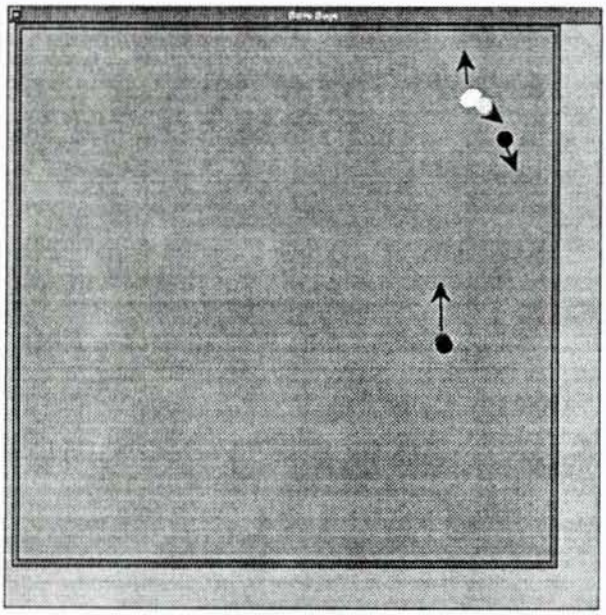


Figure 8. The maneuver is successful. A white agent is lured away, and becomes timid just before vanishing.

Further investigation is needed to determine how realistic this behavioral model really is and how best to incorporate the model into realistic environments like SIMNET. The model contains much scope for generalization, the potentials could be multi-valued (e.g. vector or tensor values) to produce more complicated motions and to include the possibility of modeling the effects of terrain and other environmental influences.

BIBLIOGRAPHY

Gilmore, Robert. Catastrophe Theory for Scientists and Engineers. New York: John Wiley and Sons, 1981.

Poundstone, William. The Recursive Universe. New York: William Morrow and Company, Inc., 1985.

Rabiner, Lawrence R., and Bernard Gold. Theory and Application of Digital Signal Processing. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1975.

Zeeman, E.C. "Catastrophe Theory." Scientific American April 1976: 65-83.

Appendix 2

Source Code

Makefile - Unix makefile

```
#
# Application makefile.
#

#
# Name of the application.
#
NAME = Bugs

#
# Source files for this application.
#
MFILES =
HFILES =
FFILES = Bugs.f          Bugs_subs.f     Bugs_disp.f                    Timer.f 2D_FFT.f  Sub-FFT.f\
        bitrev.f
CFILES =
NIBFILES =
TIFFFILES =
PSWFILES =
SNDFILES =

#
# Libraries used by this application.
#
LIBS = -lNeXT_s -lsys_s

#
# Flags to pass on to the compiler and linker.
#
CFLAGS =
FFLAGS = -object -K
LDFLAGS= -segcreate __ICON __header emptyfile -segcreate __ICON __tiff emptyfile
#
# Rules.
#
SRCFILES = $(MFILES) $(HFILES) $(CFILES) $(FFILES) $(NIBFILES) $(TIFFFILES) $(PSWFILES)
OBJFILES = $(MFILES:.m=.o) $(CFILES:.c=.o) $(FFILES:.f=.o) $(PSWFILES:.psw=.o)
DERIVED = $(PSWFILES:.psw=.c)
GARBAGE = $(DERIVED) core errs emptyfile

$(NAME): $(OBJFILES)
        touch emptyfile
```

```
	$(FC) -o  $@ $(FFLAGS) $(OBJFILES) $(LIBS) $(LDFLAGS)

clean:
	-rm -f *.o $(NAME) $(DERIVED) $(GARBAGE)

help:
	@echo '  make $(NAME) - to make the application'
	@echo '  make clean -    to remove all files but the source'
```

Bugs.inc – Variable Declarations

```
      integer nx,nxm,meanx,meany,npoten,ngen,nodisp,liveb,livef
      logical stopped
      real xmax,ymax,x0,y0,dx,dy
      parameter (nx=32,nxm=nx-1)
      parameter (ncomb=200)
      real budr(0:nxm,0:nxm),budi(0:nxm,0:nxm)
      real foer(0:nxm,0:nxm),foei(0:nxm,0:nxm)
      real wbudr(0:nxm,0:nxm),wbudi(0:nxm,0:nxm)
      real wfoer(0:nxm,0:nxm),wfoei(0:nxm,0:nxm)
      real xb(ncomb),yb(ncomb)
+  ,stateb(ncomb),budbud(ncomb),budfoe(ncomb)
      real xf(ncomb),yf(ncomb)
+  ,statef(ncomb),foebud(ncomb),foefoe(ncomb)
      common /GLOBALS/xmax,ymax,x0,y0,dx,dy,meanx,meany,
+ stopped,emax,nbud,nfoe,npoten,ngen,nodisp,liveb,livef,
+ budr,budi,foer,foei,wbudr,wbudi,wfoer,wfoei,
+ xb,yb,stateb,budbud,budfoe,
+ xf,yf,statef,foebud,foefoe
```

Bugs.f – Main Program

```
!       Bipolar Bugs program written in FORTRAN and compiled
!       on NeXT with Object-Oriented FORTRAN from Absoft
!
!       The coordinate system of the plot is flipped so that coordinate (0,0)
!       is in the upper-left corner of the view.

    INCLUDE "appkit.inc"                ! Kit constants
        INCLUDE "Timer.inc"

    @INTERFACE Bugs : View
c
        INCLUDE "Bugs.inc"
c



    @+ newView:REAL*4 r(4)
        @- suspend
        @- gofast
        @- step
        @- density
c       @- mouseDown:(NXEvent *)event
    @- drawSelf:REAL*4 rect(4) :INTEGER dummy
        @END

    @IMPLEMENTATION Bugs : View

    @+ newView:REAL*4 r(4)              ! Create a new view in window

        stopped=.false.
        npoten=0
        nodisp=1
         CALL NXEraseRect(rect)
     CALL PSsetgray(val(0.0))
         x0=10.
         y0=10.
         xmax=720;dx=aint((xmax-x0)/nxm)
         ymax=720;dy=aint((ymax-y0)/nxm)
         xmax=2+nxm*dx;ymax=2+nxm*dy
         idx=(700-xmax)/2;idy=(700-xmax)/2
         xmax=xmax+idx;ymax=ymax+idy
         x0=x0+idx;yo=yo+idy
         print *,"dx= ",dx," dy= ",dy
```

```
                x0=x0+1;y0=y0+1
                meanx=(nxm)/2;meany=(nxm)/2
        print *,"Initializing"
         call initialize
         print *,"Displaying"
         call display

    self = [self newFrame:&r]

     [Timer newTimer:  @0.02D0
    +       target:    self
    +       action:    Selector("display\0")]


c       [self setFlip:YES]
    newView_ = self
    @END

        @- step                         ! Suspend evolution and do a single step
    stopped = .false.                   ! Temporarily turn off suspension
    [self display]                  ! Display new grid
    stopped = .true.                    ! Suspend evolution
    step = self                         ! Return, by convention, self
    @end

        @- suspend                              ! Suspend evolution of grid
        stopped = .not. stopped         ! Toggle suspension state
        suspend = self                  ! Return, by convention, self
        @end

        @- gofast                               ! Suspend display of cells
        nodisp = 1-nodisp                       ! Toggle grid display
        gofast = self                   ! Return, by convention, self
        @end

        @- density                              ! Toggle display of background density
        npoten=npoten+1
        if (npoten.gt.4) npoten=0
        density = self                  ! Return, by convention, self
        @end

c       @- mouseDown
c*
c* This method handles a mouse down.
c*
c       mouseDown=self
```

```
c       @end

    @- drawSelf:REAL*4 rect(4) :INTEGER count
        ! Draw window view

    ! Calculate and draw graph

        if(stopped) return
c repeat
c10     continue
c       print *,"Summing neighbors"
        call neighbor
c       print *,"Applying rules"
        call rule(xb,yb,emax,
   +  stateb,budfoe,budbud,foei,budr,nxm,nbud)
        call rule(xf,yf,emax,
   +  statef,foebud,foefoe,budi,foer,nxm,nfoe)
c       print *,"Displaying"
        call display
c until keypressed;
c       go to 10

        drawSelf__ = self

    @END


    @end                                ! End of implementation

        PROGRAM main
    IMPLICIT NONE
    INTEGER NXApp                       ! Our application object
    INCLUDE "object.inc"                ! Object-Oriented FORTRAN definitions

    NXApp = [Application new]           ! Get ID of new Application object

    CALL setUp()                        ! Set up the environment
    [NXApp run]                 ! Start the event loop for application
    [NXApp free]                        ! Free program space
    END


    SUBROUTINE setUp                            ! Routine to set up the environment
    IMPLICIT NONE
    INTEGER    myWindow, myPanel, myMenu, panelText, myView
        INTEGER         content
    REAL*4      rect(4), viewRect(4)
```

```
INCLUDE "object.inc"              ! Object-Oriented FORTRAN definitions
include "appobject.inc"           ! Application kit constant definitions


! Set up a Window
CALL NXSetRect(rect,val(100.0),val(100.0),val(800.0),val(800.0))
myWindow = [Window newContent:  &rect
+               style:          NX_TITLEDSTYLE
+               backing:        NX_BUFFERED
+               buttonMask:     NX_MINIATURIZEBUTTONMASK
+               defer:          NO]
[myWindow setTitle:"Battle Bugs\0"]
[[myWindow contentView] getBounds:&viewRect]
myView = [Bugs newView:&viewRect]
     content = [myWindow contentView]
[[myWindow contentView] addSubview:myView]

! Set up an information Panel
CALL NXSetRect(rect,val(400.0),val(800.0),val(255.0),val(60.0))
myPanel = [Panel newContent:      &rect
+               style:          NX_TITLEDSTYLE
+               backing:        NX_BUFFERED
+               buttonMask:     NX_CLOSEBUTTONMASK
+               defer:          YES]
[myPanel setTitle:"About Bugs\0"]
[myPanel removeFromEventMask:(NX_KEYDOWNMASK .or. NX_KEYUPMASK)]

CALL NXSetRect(rect,val(0.0),val(0.0),val(255.0),val(55.0))
panelText = [Text newFrame:       &rect
+               text:           "Battle Bugs is written in
+FORTRAN using Object-Oriented FORTRAN (TM) from
+Absoft Corporation.  T. L. Clarke\0"
+               alignment:      NX_CENTERED]
[panelText setSelectable:NO]
[[myPanel contentView] addSubview:panelText]

! Set up a Menu
myMenu = [Menu newTitle:"Bugs\0"]
[[myMenu addItem:               "Info...\0"
+       action:                 Selector("orderFront:\0")
+       keyEquivalent:          NULL] setTarget:myPanel]
[[myMenu addItem:               "Pause\0"
+       action:         Selector("suspend\0")
+       keyEquivalent: ichar('p')] setTarget:myView]
[[myMenu addItem:               "GoFast\0"
+       action:         Selector("gofast\0")
```

```
+       keyEquivalent: ichar('f')] setTarget:myView]
 [[myMenu addItem:              "Single Step\0"
+       action:          Selector("step\0")
+       keyEquivalent: ichar('s')] setTarget:myView]
 [[myMenu addItem:              "Show Density\0"
+       action:          Selector("density\0")
+       keyEquivalent: ichar('q')] setTarget:myView]
 [[myMenu addItem:              "Print...\0"
+       action:          Selector("smartPrintPSCode:\0")
+       keyEquivalent: ichar('D')] setTarget:myWindow]
 [myMenu addItem:               "Hide\0"
+       action:          Selector("hide:\0")
+       keyEquivalent: ichar('h')]
 [myMenu addItem:               "Quit\0"
+       action:          Selector("terminate:\0")
+       keyEquivalent: ichar('q')]
 [myMenu sizeToFit]
 [GetNXApp() setMainMenu:myMenu]

 ! Display all three windows and PostScript text in view
 [myPanel display]
 [myMenu display]
 [myWindow display] ! Calculate and display graph

 ! Move myWindow on-screen
 [myWindow orderFront:nil]
 END
```

c

Bugs_disp.f – Display routines

```
c

      Subroutine Box(xlow,xtop,ylow,ytop,grey)
c     call PSnewpath()
      call PSmoveto(VAL(xlow),VAL(ylow))
      call PSlineto(VAL(xtop),VAL(ylow))
      call PSlineto(VAL(xtop),VAL(ytop))
      call PSlineto(VAL(xlow),VAL(ytop))
      call PSlineto(VAL(xlow),VAL(ylow))
      call PSclosepath()
      call PSsetgray(val(grey))
      call PSfill()
      call PSstroke()
      call PSmoveto(VAL(xlow),VAL(ylow))
      call PSlineto(VAL(xtop),VAL(ylow))
      call PSlineto(VAL(xtop),VAL(ytop))
      call PSlineto(VAL(xlow),VAL(ytop))
      call PSlineto(VAL(xlow),VAL(ylow))
      call PSsetgray(val(0.0))
      call PSstroke()
c     call PScopypage()
      return
      END

c Battle Bug Subroutines

      subroutine display
      INCLUDE "Bugs.inc"
      integer i,nit,living(100)
      real xx0,r
      r=dx/2
      xx0=x0+dx/2
      ngen=ngen+1
      nit=max0(nbud,nfoe)
      call Box (0.0,800.,ymax+2*dy+9,800.,0.666667)
      if ((nodisp.ne.0) .or. (mod(ngen,10).eq.0) ) then
c
      call Box(x0-10,xmax+2*dx+8,y0-10,ymax+2*dy+8,0.333333)
      call Box(x0-2,xmax+2*dx,y0-2,ymax+2*dy,0.5)
c     print *,"nbud, nfoe ",nbud,nfoe

c  code displays potential function
      if (npoten.ne.0) then
c     print *,"# potential displayed ",npoten
```

```
      pmax=0.0;pmin=1.0e6
      if (npoten.eq.1) then
            do i=0,nxm;do j=0,nxm
            pmax=max(pmax,(budr(i,j)))
            pmin=min(pmin,(budr(i,j)))
            end do; end do
      end if
      if (npoten.eq.2) then
            do i=0,nxm;do j=0,nxm
            pmin=min(pmin,(budi(i,j)))
            pmax=max(pmax,(budi(i,j)))
            end do; end do
      end if
      if (npoten.eq.3) then
            do i=0,nxm;do j=0,nxm
            pmin=min(pmin,(foer(i,j)))
            pmax=max(pmax,(foer(i,j)))
            end do; end do
      end if
      if (npoten.eq.4) then
            do i=0,nxm;do j=0,nxm
            pmin=min(pmin,(foei(i,j)))
            pmax=max(pmax,(foei(i,j)))
            end do; end do
      end if

c     print *,"max/min potential ",pmax,pmin
      ascale=1.0/(pmax-pmin)
      do i=0,nxm
      x=xx0+dx*(i)
c     print *,"x= ",x," y= ",y
      do j=0,nxm
      y=y0+dy*(j)
            call PSsetlinewidth(val(dx))
            if (npoten.eq.1) dens=budr(i,j)
            if (npoten.eq.2) dens=budi(i,j)
            if (npoten.eq.3) dens=foer(i,j)
            if (npoten.eq.4) dens=foei(i,j)
            CALL PSsetgray(val((dens-pmin)*ascale))
            call PSnewpath()
      CALL PSmoveto(val(x),val(y))
            call PSlineto(val(x),val(y+dy))
            call PSstroke
      end do;
      end do;
      call PSsetlinewidth(val(1.0))
c
```

```
        call PSsetgray(val(0.0))
        do i=1,100;living(i)=0;end do
        encode(9,301,living)
        call PSselectfont(living,VAL(16.0))
        call PSnewpath()
        call PSmoveto(VAL(x0),VAL(ymax+dy))
c
        do i=1,100;living(i)=0;end do
        if (npoten.eq.1) encode(13,201,living)
201     format("Friend-Friend")
        if (npoten.eq.2) encode(13,202,living)
202     format("Friend-Foe   ")
        if (npoten.eq.3) encode(13,203,living)
203     format("Foe-Foe      ")
        if (npoten.eq.4) encode(13,204,living)
204     format("Foe-Friend   ")
        call PSshow(living)
c
        call PSsetgray(val(0.0))
        call PSstroke()
        call PScopypage()
c
        end if


        liveb=0
        livef=0
        do i=1,nit
        if (i.le.nbud) then
          if (stateb(i).gt.0.0) then
            liveb=liveb+1
            x=dx*(xb(i)+1);y=dy*(yb(i)+1)
c       print *,"x= ",x," y= ",y
            grey=0.9
            if(stateb(i).gt.1.0) grey=1.0
            CALL PSsetgray(val(grey))
            call PSnewpath()
        CALL PSmoveto(val(x),val(y))
            call PSarc(val(x),val(y),val(r),val(0.),val(360.))
            call PSfill()
            call PSstroke()
            end if
          end if
        grey=0.0
        if (i.le.nfoe) then
          if (statef(i).gt.0.0) then
```

```
            livef=livef+1
            x=dx*(xf(i)+1);y=dy*(yf(i)+1)
c           print *,"x= ",x," y= ",y
            grey=0.1
            if(stateb(i).gt.1.0) grey=0.0
            CALL PSsetgray(val(grey))
            call PSnewpath()
          CALL PSmoveto(val(x),val(y))
            call PSarc(val(x),val(y),val(r),val(0.),val(360.))
            call PSfill()
            call PSstroke
            end if
           end if
          end do
c
          end if
          CALL PSsetgray(val(0.0))
          do i=1,100;living(i)=0;end do
          encode(9,301,living)
301       format("Helvetica")
          call PSselectfont(living,VAL(16.0))
          call PSmoveto(VAL(x0),VAL(ymax+3*dy))
          do i=1,100;living(i)=0;end do
          encode(20,101,living) liveb
101       format("Surviving white: ",i3)
          call PSshow(living)

          call PSmoveto(VAL(3.0*(x0+xmax)/4.0),VAL(ymax+3*dy))
          do i=1,100;living(i)=0;end do
          encode(20,103,living) livef
103       format("Surviving black: ",i3)
          call PSshow(living)
          call PSmoveto(VAL(2.0*(x0+xmax)/4.0),VAL(ymax+4*dy))
          do i=1,100;living(i)=0;end do
          encode(22,105,living) ngen
105       format("Elapsed Cycles: ",i6)
          call PSshow(living)
c
          call PSstroke()
c         call PScopypage()
c         call PSsetlinewidth(val(1.0))

          return
          END
```

```
      subroutine neighbor
      INCLUDE "Bugs.inc"

c wr,wi is transformed weight array
c far,fai is used to for calculations
c
c a is active grid, b gets neighbor count
c
      integer i,j,nxmm,nxmp
c     if (nbud.gt.0) return
      nxmp=nxm+1
      nxmm=nxm-1
      do i=0,nxm;do j=0,nxm
            budr(i,j)=0.0;budi(i,j)=0.0
            foer(i,j)=0.0;foei(i,j)=0.0
      end do;
      end do
      do i=1,nbud
            ix=mod(nx+nint(xb(i)),nx)
            iy=mod(nx+nint(yb(i)),nx)
            budr(ix,iy)=budr(ix,iy)+1.0
      end do
      do i=1,nfoe
            ix=mod(nx+nint(xf(i)),nx)
            iy=mod(nx+nint(yf(i)),nx)
            foer(ix,iy)=foer(ix,iy)+1.0
      end do

      call FFT2DREV(budr,budi,nxmp,-1,0)
      call FFT2DREV(foer,foei,nxmp,-1,0)
c     print *,"returned form FFT2DREV"
c multiply by complex conjugate
      fmax=-1.0e20;bmax=-1.0e20
      fmin=-fmax;bmin=-bmax
      do i=0,nxm;do j=0,nxm
            tr=budr(i,j)*wbudr(i,j)-budi(i,j)*wbudi(i,j)
            ti=budi(i,j)*wbudr(i,j)+budr(i,j)*wbudi(i,j)
            budr(i,j)=tr;budi(i,j)=ti
      end do;end do
      do i=0,nxm;do j=0,nxm
            tr=foer(i,j)*wfoer(i,j)-foei(i,j)*wfoei(i,j)
```

```
                ti=foei(i,j)*wfoer(i,j)+foer(i,j)*wfoei(i,j)
                foer(i,j)=tr;foei(i,j)=ti
        end do;end do
c       print *,"calling REVFFT2D"
        call REVFFT2D(budr,budi,nxmp,+1,0)
        call REVFFT2D(foer,foei,nxmp,+1,0)
c       do i=0,nxm;do j=0,nxm
c               fmax=max1(fmax,foer(i,j))
c               bmax=max1(fmax,budr(i,j))
c               fmin=min1(fmin,foer(i,j))
c               bmin=min1(fmin,budr(i,j))
c       end do; end do
c       print *,"budmax, budmin, foemax, foemin ",
c  + bmax,bmin,fmax,fmin

        return
        END

        subroutine initialize
        INCLUDE "Bugs.inc"
        integer i,j,nxmp
        real weight1,weight2
        ngen=0
        nxmp=nxm+1
c initialize weight array and transform
        smax=(nxm+1);xfold=smax/2
        emax=nxm
        sig1=smax/4.0
        as1=1/sig1**2/2.0;an1=1/(6.2832*sig1)/smax**2
        sig2=amin1(smax/8.0,1.0)
        as2=1/sig2**2/2.0;an2=1/(6.2832*sig2)/smax**2
        sig3=smax/4.0
        as3=1/sig3**2/2.0;an3=1/(6.2832*sig3)/smax**2
        print *,"sig1, sig2, sig3 ",sig1,sig2,sig3
c       an2=3.0
        hywt=1./smax**2
        do i=0,nxm;do j=0,nxm
                ddx=i;if (ddx.gt.xfold) ddx=ddx-smax
                ddy=j;if (ddy.gt.xfold) ddy=ddy-smax
                arg=ddx**2+ddy**2
                weight1=an3*exp(-arg*as3)-an2*exp(-arg*as2)
                weight2=an1*exp(-arg*as1)
c               weight=hywt/amax1(1.0,arg)-an2*exp(-arg*as2)
c               weight=hywt/amax1(1.0,arg)
c note use of real part for frienc (bud) and imaginary part for foe
                wbudr(i,j)=weight1
```

```
                wbudi(i,j)=weight2
                wfoer(i,j)=weight1
                wfoei(i,j)=weight2
        end do;end do
c don't count yourself ?
c       wbudr(0,0)=0.0;wbudi(0,0)=0
c       wfoer(0,0)=0.0;wfoei(0,0)=0
        call FFT2DREV(wbudr,wbudi,nxmp,-1,0)
        call FFT2DREV(wfoer,wfoei,nxmp,-1,0)

c psuedo random function y=(991*y mod 1024) + 31
        iseed = 59
        do i=1,ncomb
                iseed=mod(iseed*157,1024)+31
                xb(i)=smax*iseed/3240.+xfold
                iseed=mod(iseed*97,1024)+31
                yb(i)=smax*iseed/3240.+xfold
                stateb(i)=0.5
        end do
        nbud=ncomb/2
        nbud=3
        do i=1,ncomb
                iseed=mod(iseed*157,1024)+31
                xf(i)=smax*iseed/3240.0
                iseed=mod(iseed*97,1024)+31
                yf(i)=smax*iseed/3240.0
                statef(i)=0.5
        end do
        nfoe=ncomb/2
        nfoe=3
c  need to call neighbor to initialize the (bud,foe)(bud,foe) arrays
c  when full behaviour is implemented

        return
        END

        subroutine rule
c code is written from viewpoint of foe
   + (xf,yf,emax,statef,foebud,foefoe,bud,foe,nxm,nfoe)
        real bud(0:nxm,0:nxm),foe(0:nxm,0:nxm)
        real xf(nfoe),yf(nfoe)
   +  ,statef(nfoe),foebud(nfoe),foefoe(nfoe)
        integer i,nx
        nx=nxm+1
        ip=1;im=-1
        foeequ=5.0
        swfact=3.0
```

```
c       dead=20.0
        dead=8.1175
        do i=1,nfoe
                ix=mod(nx+nint(xf(i)),nxm)
                iy=mod(nx+nint(yf(i)),nxm)
c mode switching hysteresis - catastrophe approx -
c with switching along lines foe=2*bud and bud=2*foe
        bud0=abs(bud(ix,iy))
        foe0=abs(foe(ix,iy))
c if overwhelmed kill it
        if (bud0.gt.(dead*foe0)) statef(i)=0.0
c check if alive
        if (statef(i).gt.0.0) then
         if (bud0.gt.(swfact*foe0)) statef(i)=0.5
         if (foe0.gt.(swfact*bud0)) statef(i)=2.0
c  set motion increments
         if (statef(i).gt.1.0) then
              v1=0.25;v2=1.0
               else
                    v1=0.5;v2=-0.10
               endif
               gradbx=bud(mod(ix+ip,nx),iy)-bud(mod(nx+ix+im,nx),iy)
               gradby=bud(ix,mod(iy+ip,nx))-bud(ix,mod(nx+iy+im,nx))
               gradfx=foe(mod(ix+ip,nx),iy)-foe(mod(nx+ix+im,nx),iy)
               gradfy=foe(ix,mod(iy+ip,nx))-foe(ix,mod(nx+iy+im,nx))
c
               if (gradfx.gt.0) then
                    xf(i)=xf(i)+v1
               else
                    xf(i)=xf(i)-v1
               end if
               if (gradbx.gt.0) then
                    xf(i)=xf(i)+v2
               else
                    xf(i)=xf(i)-v2
               end if
               if (xf(i).gt.emax) xf(i)=xf(i)-emax
               if (xf(i).lt.0.0) xf(i)=xf(i)+emax
c
               if (gradfy.gt.0) then
                    yf(i)=yf(i)+v1
               else
                    yf(i)=yf(i)-v1
               end if
               if (gradby.gt.0) then
                    yf(i)=yf(i)-v2
```

```
        else
                yf(i)=yf(i)+v2
        end if
        if (yf(i).gt.emax) yf(i)=yf(i)-emax
        if (yf(i).lt.0.0) yf(i)=yf(i)+emax
end if          ! end of kill check
end do



return
END
```

2D_FFT.f – Two Dimensional FFT

```
c  transform square nxn (power or 2) matrix of values
c   if isw=0 leave result unstransposed and don't perfrom bitreversals
c   in order to speed correlation calculations
        subroutine FFT2DREV(ar,ai,n,is,isw)
        dimension ar(n,n),ai(n,n)
c transform rows
        do i=1,n
        call FFTREV(ar(1,i),ai(1,i),n,is)
        if (isw.ne.0) call BITREV(ar(1,i),ai(1,i),n)
        end do
c transpose
        call transpose(ar,ai,n)
c transform columns
        do i=1,n
        call FFTREV(ar(1,i),ai(1,i),n,is)
        if (isw.ne.0) call BITREV(ar(1,i),ai(1,i),n)
        end do
c transpose back if desired
        if (isw.ne.0) call transpose(ar,ai,n)
        return
        end
c
c  transform square nxn (power or 2) matrix of values
c   if isw=0 assume input unstransposed and un-bitreversed
c   in order to speed correlation calculations
        subroutine c(ar,ai,n,is,isw)
        dimension ar(n,n),ai(n,n)
c transform rows
        do i=1,n
c transpose if needed
        if (isw.ne.0) call transpose(ar,ai,n)
c bit reverse if needed
        if (isw.ne.0) call BITREV(ar(1,i),ai(1,i),n)
        call      REVFFT(ar(1,i),ai(1,i),n,is)
        end do
c transpose
        call transpose(ar,ai,n)
c transform columns
        do i=1,n
c bit reverse if needed
        if (isw.ne.0) call BITREV(ar(1,i),ai(1,i),n)
        call REVFFT(ar(1,i),ai(1,i),n,is)
```

```
      end do
      return
      end
c
c transpose nxn square matrices
      subroutine transpose(A,B,N)
      real A(N,N),B(N,N)
      do i=1,(N-1)
      do j=(i+1),N
            T=A(i,j);A(i,j)=A(j,i);A(j,i)=T
            T=B(i,j);B(i,j)=B(j,i);B(j,i)=T
      end do
      end do
      return
      end
```

Sub-FFT.f – One dimensional FFT

```
c
c algorithm for FFT with bit reversed output
c
c
      SUBROUTINE FFTREV(XREAL,XIMAG,nn,is)
      DIMENSION XREAL(1:nn), XIMAG(1:nn)
      INTEGER N,NU,N2,dk,dN,dk2
      real dwr,dwi,dthet,wr,wi
      N=nn
      NU=ilog2(N)
      N2 = N/2
      K = 0
C
C
c loop over log2N sub FFTs
c
      DO  L = 1,NU
c input increment to SubFFT
      dN=2**(L-1)
c length of SubFFT-1
      dk=2**(NU-L)
      dk2=dk*2
      dthet=6.283185307*float(dN*is)/float(N)
      dwr=cos(dthet); dwi=sin(dthet)
      k=1
c     print *,"L,dN,dk ",L,dN,dk
c loop over
      wr=1.0;wi=0.0
      DO  I = 1,dk
            do m=1,N,dk2
                  k=I+m-1
c     print *,"I,m,k,k+dk",I,m,k,k+dk
                  x1r=XREAL(k);x1i=XIMAG(k)
                  x2r=XREAL(k+dk);x2i=XIMAG(k+dk)
                  dr=x1r-x2r;di=x1i-x2i
                  XREAL(k)=x1r+x2r;XIMAG(k)=x1i+x2i
                  XREAL(k+dk)=dr*wr-di*wi
                  XIMAG(k+dk)=dr*wi+di*wr
                  end do
            if (dk.ne.1) then
                  tr=wr*dwr-wi*dwi
                  wi=dwr*wi+dwi*wr
```

```
                        wr=tr
                end if
                end do
        end do
C
C
        RETURN
        END
c
c algorithm for FFT with bit reversed input
c
c
        SUBROUTINE REVFFT(XREAL,XIMAG,nn,is)
        DIMENSION XREAL(1), XIMAG(1)
        INTEGER N,NU,N2,dk,dN,dk2
        real dwr,dwi,dthet,wr,wi
        N=nn
        NU=ilog2(N)
        N2 = N/2
        K = 0
C
C
c loop over log2N sub FFTs
c
        DO  L = NU,1,-1
c input increment to SubFFT
        dN=2**(L-1)
        dthet=6.283185307*float(dN*is)/float(N)
        dwr=cos(dthet); dwi=sin(dthet)
c length of SubFFT-1
        dk=2**(NU-L)
        dk2=dk*2
        k=1
c       print *,"L,dN,dk ",L,dN,dk
c loop over
        wr=1.0;wi=0.0
        DO  I = 1,dk
                do m=1,N,dk2
                        k=I+m-1
c       print *,"I,m,k,k+dk",I,m,k,k+dk
                        x1r=XREAL(k);x1i=XIMAG(k)
                        x2r=XREAL(k+dk);x2i=XIMAG(k+dk)
                        dr=x2r*wr-x2i*wi;di=x2i*wr+x2r*wi
                        XREAL(k)=x1r+dr;XIMAG(k)=x1i+di
                        XREAL(k+dk)=x1r-dr
                        XIMAG(k+dk)=x1i-di
```

```
                end do
            if (dk.ne.1) then
                tr=wr*dwr-wi*dwi
                wi=dwr*wi+dwi*wr
                wr=tr
            end if
            end do
        end do
C
C
        RETURN
        END
```

bitrev.f – Reorders Time Series

```
 C Bit Reversal subroutine for complex data from FFT
c
        SUBROUTINE BITREV(XREAL,XIMAG,n)
        DIMENSION XREAL(0:n), XIMAG(0:n)
        integer K,NU,m
        m=n
        NU=ilog2(m)
C
        DO 103 K = 1,m
          I = IBITR(K ,NU)
          IF (I .LE. K) GO TO 103
          TREAL = XREAL(K)
          TIMAG = XIMAG(K)
          XREAL(K) = XREAL(I)
          XIMAG(K) = XIMAG(I)
          XREAL(I) = TREAL
          XIMAG(I) = TIMAG
103     CONTINUE
C
C
        RETURN
        END
c
        FUNCTION IBITR(J,NU)
        INTEGER J1,I,J2,JBITR
        J1 = J
        JBITR = 0
C
C
        DO 200 I = 1, NU
        J2 = J1/2
        JBITR = JBITR * 2 + (J1 - 2 * J2)
200     J1 = J2
C
C
        IBITR=JBITR
        RETURN
        END
c
        function ilog2(n)
        ilog2=int(1.442695041*log(float(n))+.01)
c       print *,"n, log2n ",n,ilog2
        return
```

```
end
```

Timer.f – Supplied by Absoft

```
! Timer class for real-time control
!
! Start timer with newTimer:target:action: method
! Stop timer with freeTimer: method
    INCLUDE "Object.inc"                                    ! Object class definition
    INCLUDE "Timer.inc"

! TimerFunc is the FORTRAN function invoked at each time interval

    SUBROUTINE TimerFunc(dum1, dum2, dum3)
    INTEGER*4                                               theTarget
    INTEGER*4                                               theAction
    COMMON /TIMER_GLOBALS/ theTarget, theAction
     INCLUDE "object.inc"

    [theTarget perform:theAction]
    END

    @implementation Timer : Object

! newTimer:target:action: method starts a timer that causes the 'action' method
!
      to be invoked in the 'target' at each 'interval'.
! Returns the ID of the timer entry.

    @+ newTimer:REAL*8 interval target:target action:action
    EXTERNAL                                                TimerFunc
    INCLUDE                                                 "ApplicationPARAM.inc"

    theTarget = target
    theAction = action
    self = DPSAddTimedEntry(@interval,VAL(LOC(TimerFunc)),VAL(self),
   +VAL(NX_MODALRESPTHRESHOLD+5))
    @end

! freeTimer: removes the timed entry for the specified timer ID

    @- freeTimer:integer entry
    CALL DPSRemoveTimedEntry(VAL(entry))
    freeTimer = self
    @end
```

```
@end
```