Modeling Hostage Situations

Manzano High School

Team 60:

Eric Benner
Steven Benner
Samuel Bonin
Haley McDuff
Anthony Rodriguez

Sponsor:

Stephen Schum

Mentor:

Paul Roensch

Table of Contents:

Introduction:

Hostage situations are complex scenarios involving many complicating factors. We used our own mathematical equations to hypothetically model hostage negotiations. The negotiation phase of a hostage situation is the most critical part overall to law enforcement agencies. We used the equations to model scenarios, decisions, and outcomes involved in a hostage crisis. We used C++ and Microsoft Excel to aid us with our model.

Executive Summary:

Every year there are many hostage situations in the United States and the world. These come in many forms, ranging from a subject barricading only himself to multiple hostage takers with multiple hostages. This alone makes the situations intensely complex for hostage negotiators. Furthermore, by adding the human element, which includes psychology and emotion, the situations become even more complex. Negotiators go through extensive training to prepare for the many scenarios they might face. However, no two hostage negotiators handle the same crisis in the same way, resulting in seemingly infinite possibilities.

Original Ideas:

Originally, we intended for our project to be game theory applied to the board game Risk. We decided we could find no practical real world application for this project; consequently, we changed our project shortly after the kickoff conference. We decided hostage situations would provide an ideal setup for a game-theory based program, since we believed they involved definite and predictable amounts of negotiation. Little did we know how wrong we where and how hard this project would be! So to narrow our field, we also decided to lessen the emphasis on game theory.

1st Program:

Our first program turned out to be more of a probability simulation (and stepping stone) than a functioning model of game theory in hostage situations. The intent was to output data showing the steps taken in a hostage situation and its resolution. Each action, determined by certain probabilities, would affect the next player's action. The first action would be taken by the hostage negotiator. This alternating series of decisions would ultimately result in one of six outcomes. During this time, we completed a bit of research and were able to have several interviews with Law Enforcement personnel. We had a functioning program, albeit drastically unrealistic, and it had the ability to be easily modified for further realism. However, we soon revamped the program with the comparatively recent arrival of team member Haley McDuff, who showed us the error of our ways.

2nd Program:

We realized that our program, while functioning, was difficult to compare to real world scenarios, and we still had few interesting mathematical equations to use. Consequently, Haley suggested making a program that would output a single decision tree and which would show all possible situations with the set of input data, instead of merely one series of events. In this way, we would be able to take real-world events and compare them to what our output tree would be. In the first case we decided to model the hostage situation in Beslan, Russia (highly modified due to scarcity of information). If one branch, so to speak, were to match the decisions and outcomes of Beslan, we could then move on to model further hostage crises. We worked with Beslan because it had definite beginning variables and data. After we determined that Beslan has been accurately modeled, we intended to model the hostage situation that occurred during the 1972 Olympic Games in Munich, Germany.

Research:

We obtained our information from three types of sources: the internet, personal interviews, and books. We began making lists of data from hostage situations we found online. However, this source of information did not have enough details for our project. We did extensive research into the hostage situation that occurred on September 1, 2004 in Beslan, Russia. We also heavily researched the hostage incident that occurred at the 1972 Olympic Games in Munich, Germany.

Our first interview was with the former Albuquerque Police Department Chief Negotiator of the Crisis Negotiation Team. He was able to give us insight into how a crisis negotiator approaches hostage situations. He also talked to us about hostage situations and referred us to the FBI. Our second interview was with Special Agent Ray Gonzales from the Albuquerque Field Office of the FBI. He is the agent in charge of the Albuquerque FBI SWAT team. Mr. Gonzales was able to give us a brief overview of the FBI's hostage negotiation and rescue teams in the United States. He provided us with plenty of information concerning SWAT operations during hostage negotiation. Our third interview was with Special Agent Edward Toliver from the Albuquerque field office of the FBI. He is one of the hostage negotiators for the Crisis Negotiation Team. He talked to us about some of the techniques used to deal with a hostage taker. He provided us with a great deal of information, including a general list made by hostage takers.

Programming:

Our first program is oriented toward the collection of data. Using a classic, hypothetical hostage situation of one negotiator "playing" against one hostage taker, we decided that the hostages would be only pawns or tokens of the hostage taker to gain his demands. Similarly, the taker's demands would merely be game tokens used by the negotiator to vie for the freedom of the hostages. Both the hostage taker and the negotiator have sound mental stability, are in good health, and have the desire to live. This is evidently a gross over-simplification; the hostages were merely a number and the demands were simply put on a scale of one to ten. Also taken into account was the premise that the hostages could be killed, and also that the taker could increase or decrease time to force the hostage negotiator to accept demands. Human emotions or reasoning are non-influential in the original program.

The key to understanding the logical operation of the program is that it is not an actual game-theory based program, but it is a probability-theory program; it is partially built on ideas from game theory. The program makes "decisions" based upon probabilities. The decisions are determined by checking to see if a random number falls within a range of numbers described by a set probability. There were plans for mathematical equations that would change the probabilities based on previous actions and variables associated with different types of situations. These plans, however, were deemed unnecessary because we felt we should do more on the game theory focus of the project. The probability-changing concept merely scratched the surface of game-theory ideas.

The program runs on a time scale. The program is built to run on a loop representative of the change of an arbitrary integral time unit. Time also factors into the program and situation with the idea of "time left" before the taker initiates the end-game scenario of killing all

hostages. The hostage negotiator takes his turn when the time is represented by an odd number, and the hostage taker takes his turn when the time is an even number. The input of the variables of the scenario is logically considered the taker's turn at time zero. This is also an over-simplification as hostage situations do not actually run off equal steps of passing time; they run on "events" or moments of action such as conversations or time, relationship, and demand changes. These events change the feelings and conditions of the situation.

The lives of the hostages are factored into the program merely as game pieces. Since the probabilities are set to the same values throughout the game for simplicity, the death or release of hostages do not factor into the actions occurring after the fact unless those deaths or releases complete an endgame (when the either player decides to end the situation.) The hostages do serve the purpose of showing the success of the actions taken. The demands also serve practically the same purpose as the hostages in this program.

The display of this information is the most important aspect of the program itself. In the display of the program, we show what the values of six particular aspects of the scenario are for each period of time. We also display the fail rate. The fail rate is the measure of the extent to which the situation didn't achieve success on a scale from 0-10. It takes into account the number of hostages initially taken, released, or killed. Also, factored into the fail rate are the total demands wanted by the taker, the number of demands given to the taker, and the number of demands the taker dropped. The aspects of the scenario displayed are: the time that has passed or the time unit, the number of hostages that are presently dead, the scale level of demands that the hostage taker requires, the number of hostages still under the control of the taker, the demands the taker has received, and the time that the hostage taker still has remaining. This data also allows us to analyze which decisions generally seem to turn out the best for the situation.

The idea of a fail rate comes from the fact that once a hostage situation has begun, there can be no positive outcome to the situation. In the best-case scenario, all hostages are released or rescued unharmed, and the taker is taken into police custody. This best-case scenario still puts strain on the police and psychological distress on the hostages, and it wastes money and time. The worst-case scenario ends with all hostages, other civilian and law enforcement personnel, and the hostage taker killed or injured, and with many of the hostage taker's demands having been met.

Thus, this program is mainly a thin representation of what a scenario could look like in action. It does not sufficiently accomplish the goal of showing what the negotiator or the taker may be feeling or thinking.

The second program is focused more on simulating logic and emotions and inter-connecting these with finding the optimal course for solution to the situation. This program is built to create data comparable to a real situation. This program broadens the idea of the hostage negotiator and taker to be more team-based. In other words, the situation is that of a negotiation team, negotiating with the hostage takers. Often, it is groups or teams who face-off against each other, not individuals. Another primary component of this new program is the mathematics. As seen in the section, "Program 2 Equations," we attempt to focus on the tension in the situation and on the relationships between the two teams.

The inputs of the situation are more in-depth for this program than the previous. The previous program only asked for three inputs: demand scale, number of hostages, and time left. In this program, the demands are not a single demand scale but are arranged such that the taker can have up to twenty different demands; each of these demands has its own value and a corresponding severity to show what the power is for that type of demand. The demand values

are multiplied with their own severity, and then all demands with severities are summated. The inputting of hostages is also more complex. The initial number of children, adults, and elderly hostages; the number of the children, adult, and elderly casualties from the initialization of the situation; and the number of children, adults, and elderly killed in the initial action are all input. Children, adults, and elderly each have differing values of influence. There is no arbitrary time unit system in this program. The "steps," as we refer to them, are the events or actions that happen that change the situation. These events vary from conversations to demand exchanges, executions, and SWAT infiltrations.

The new program also contains new inputs. These are the perimeter and armament values. The fraction of the perimeter that is armed and the power of the maximum armament of the perimeter are input, and the fraction that is unarmed is calculated. Also, the program asks for the number of armed hostage takers and the power their arms have to be entered.

There are two types of outputs, each corresponding to the two different modes of operation. The three variables we display are the relationship of the taker to the negotiator, the relationship of the negotiator to the taker, and the over-all tension of the present situation. These are output along with the information of the actions previously taken.

Mathematical Equations:

When the hostage taker (taker) performs the initial action of taking hostages and stating demands, certain numerical values are gathered from analysis of the situation. Demands made are assigned values of one through ten based on the expense and difficulty of providing that demand (D); each demand is then assigned a severity value of one through ten based on the quantity demanded (SD). Next, the physical perimeter of the area in which the taker and hostages are barricaded is assigned a number one through ten according to defensive impenetrability (a standard wall being about three). These and other values are input into an equation (the input equation) in order to convert the initial rough data into a tension value. This tension value represents the uncertainty and distrust felt by both players after the initial action, and later it affects the relationships of the players. The first part of the input equation is the sum of the values of each demand multiplied by its corresponding severity. This creates the entire value of the demands made and is divided into the number of hostages taken; it creates the inverse of the bargaining value of each hostage. The less each hostage is worth, the more dispensable each is to the taker, and a greater possibility of losing a hostage all increase the tension of the situation. The second part of the equation involves injuries and death incurred during the initial action and the perimeter strength. If a large number of hostages are injured or killed, the negotiator would be more likely to forcibly initiate an end-game. The sum of injuries and deaths is divided by the perimeter value, consequently deterring the negotiator from such an action. The perimeter value is calculated by multiplying the strength of the perimeter at its strongest point by the fraction of the perimeter that has that strength. That value is then divided by the fraction of the perimeter having lesser strength; if the entire physical perimeter is of the same strength, the value is divided by one. This entire quotient is then added to the value of

force the takers have over the hostages. This is found by multiplying the number of hostage takers and the value of their personal armament (the overall power of force the takers have over the hostages), on a scale of one to ten, and then dividing that value by the value of all the hostages. The relationship value of the initial action is calculated by multiplying the tension and the severity of the initial action and dividing the product by11.5; the quotient is then multiplied by the action value which is negative one because the initial action of taking hostages and making demands is a negative action (a positive action would have a value of one). After the initial action, the negotiator has eight possible actions to take. Each of these actions is assigned a relationship value, tension value, and a severity constant. The relationship value of each action represents how positive or negative the action is in terms of the two players forming a relationship while the tension values represent how much the action will increase or decrease the overall tension of the game. The values for each action are plugged into their respective equations in order to calculate the relationship felt by each player and the tension of the situation after the action is taken. The severity constant for each action is calculated by using a variation of the tension function to find the approximate severity for an action that is later plugged into the severity function to predict a more specific severity. The severity value of the initial action is evaluated as the initial tension value divided by four because both the initial tension and the severity of the initial action take into account the same variables; one fourth of the tension produces a severity with the range from one to ten.

Relationship values for both the taker and the negotiator are calculated after every action including the initial actions. The relationship of the taker, the bond the taker feels he has with the negotiator after the negotiator has performed an action, is calculated by adding the relationship value of the most recent negotiator action to the quotient of the sum of all previous

negotiator action relationship values plus the relationship value of the initial action divided by two. The action that has the most impact in the taker's mind is the most recent negotiator action, but he still remembers all previous negotiator actions; however, the previous actions are not as influential, so they are halved to more closely simulate the memory of the taker. The relationship of the negotiator is the bond the negotiator feels he has with the taker. This is calculated by the sum of all the taker's actions divided by two. This is added to the average of all actions that the negotiator has taken. Because the negotiator attempts to lead the taker to a positive decision and does not react directly to the previous action of the taker, the negotiator regards all of the taker's actions to be of equal value.

The tension of the situation is also calculated after an action has been taken, excluding the initial action (considered to be the taking of hostages and initial demands). Current tension is found by dividing the sum of the positive tension values of all actions taken by the absolute value of the sum of all negative tension values of all actions taken; however, if either the numerator or denominator are zero due to certain actions taken, a one is plugged in so that certain values are not disregarded. As more positive actions are taken, the denominator becomes greater and the first term smaller thus decreasing the tension of the situation. This is then multiplied by the quotient of the sum of the severity values of all positive actions (meaning actions with a positive relationship value) divided by the sum of the severity values of all negative actions taken. A rough version of the tension has now been calculated, but the relationships of both players have an impact as well. The average of the taker and negotiator relationship values for the current action is then subtracted. If the relationships average a negative value, the subtraction is of a negative number which then adds the value, increasing the tension; whereas, if the relationship values average to be positive, they will be subtracted,

reducing the tension value. Finally, the entire answer is multiplied by 2.5 in order to produce tension values that are easier to evaluate.

Relationship and tension values are then used to calculate the severity of each potential action. The severity of the next action is calculated by subtracting the average of the relationship values of the previous action from the absolute value of the tension value of the previous action. The absolute value of the tension value is taken in order to produce a positive severity value. This is then added to the severity constant of the next potential action, and the sum is divided by four to produce a severity value ranging from zero to ten.

INPUT EQUATIONS

$$P = \frac{(P_{ms})(fracarm_{ms})}{fracP_{na}} + \frac{(\#T)(W_{personal})}{VH}$$

$$Ten_i = \left[\frac{\#H}{(D)(DV)} + \frac{V_x + V_k}{P} \right]$$

$$RV_{t_0} = \frac{(TenV_{t_0})(S_i)}{11.5}(-1)$$

MEDIAL EQUATIONS

$$Ten = \left[\frac{Ten^+}{|Ten^-|} * \frac{\sum(S^+ A)}{\sum(S^- A)} + (RT + RN) \right](2.5)$$

$$RT = NA_{tcnt} + \frac{\sum(NA) + RV_{t_0}}{2}$$

$$RN = \frac{\sum(TA)}{2} + \frac{\sum(NA)}{\#trms}$$

$$S = \left[\left(|Ten_c| - \frac{RNC + RTC}{2} \right) + SConst_n \right] \left(\frac{1}{4} \right)$$

Conclusions:

This project brought us to several conclusions about hostage negotiations, the first being that a hostage crisis is an incredibly complex scenario. We believe it to be all but impossible to account for all of the contingencies and possible occurrences in a hostage situation. There are just too many variables for it to be reasonably modeled. We have come to the conclusion that human psychology cannot be quantified. Even though this project attempts to model one of the most horrific and awful human practices, it has helped us see the elegance and complexity of the human mind.

Acknowledgements:

Paul Roench	Team Mentor
Stephen Schum	Team Sponsor
Sam Boling	Programming Help
Lt. Olfad	APD Crisis Negotiator
Special Agent Ray Gonzales	FBI SWAT Commander
Special Agent Edward Toliver	FBI Hostage Negotiator
Manzano High School	Providing the Class
The Adventures in Super-Computing Challenge	Providing the Opportunity

Appendix 1: Glossary of Terms for Equations

N – negotiator
The player attempting to form a positive relationship with the hostage taker in order to negotiate an endgame

T – taker (hostage taker)
The player holding and using hostages to bargain for given demands

V – value
A numerical value assigned to an action that represents relationship, tension or severity of that action.

A – action
An automated action performed by either player. It is assigned values for relationship, tension, and severity.

R – relationship
The bond that is formed between the players (hostage taker and the negotiator) throughout the game (negotiation)

RN – relationship of negotiator
The bond that the negotiator feels throughout the negotiation

RT – relationship of taker
The bond that the taker feels throughout the negotiation

Ten – tension
Used as a title for uncertainty and distrust between the two players throughout the game.

S – severity
A numerical value representing the intensity of an action taken by either player

D – demand(s)
What the Taker asks for in return for a peaceful endgame.

W – weapons
Term used to describe any weapon being used by the Taker either as a personal weapon or part of a barricade, offensive or defensive.

P – perimeter
A general term that incorporates the stability and offensive/defensive strength personal weapons and the physical perimeter surrounding the Taker and hostages

n – next
Subscript of a variable which is the value of that same variable in the next action

c – current
Subscript of a variable which is the value of that same variable in the current action

p – previous
Subscript of a variable which is the value of that same variable in the previous action

ms- max strength
Subscript of P to describe the maximum strength at a given point

fracarm- fraction armed at a given value

na - not armed
Subscript to describe the amount of P not armed

k= subscript to describe number of hostages or takers dead

x= subscript to describe number of hostages or takers casualty

i= $^{\text{subscript}}$ to describe initial values

+= superscript used to describe positive values

-= superscript used to describe negative values

rcnt= recent event

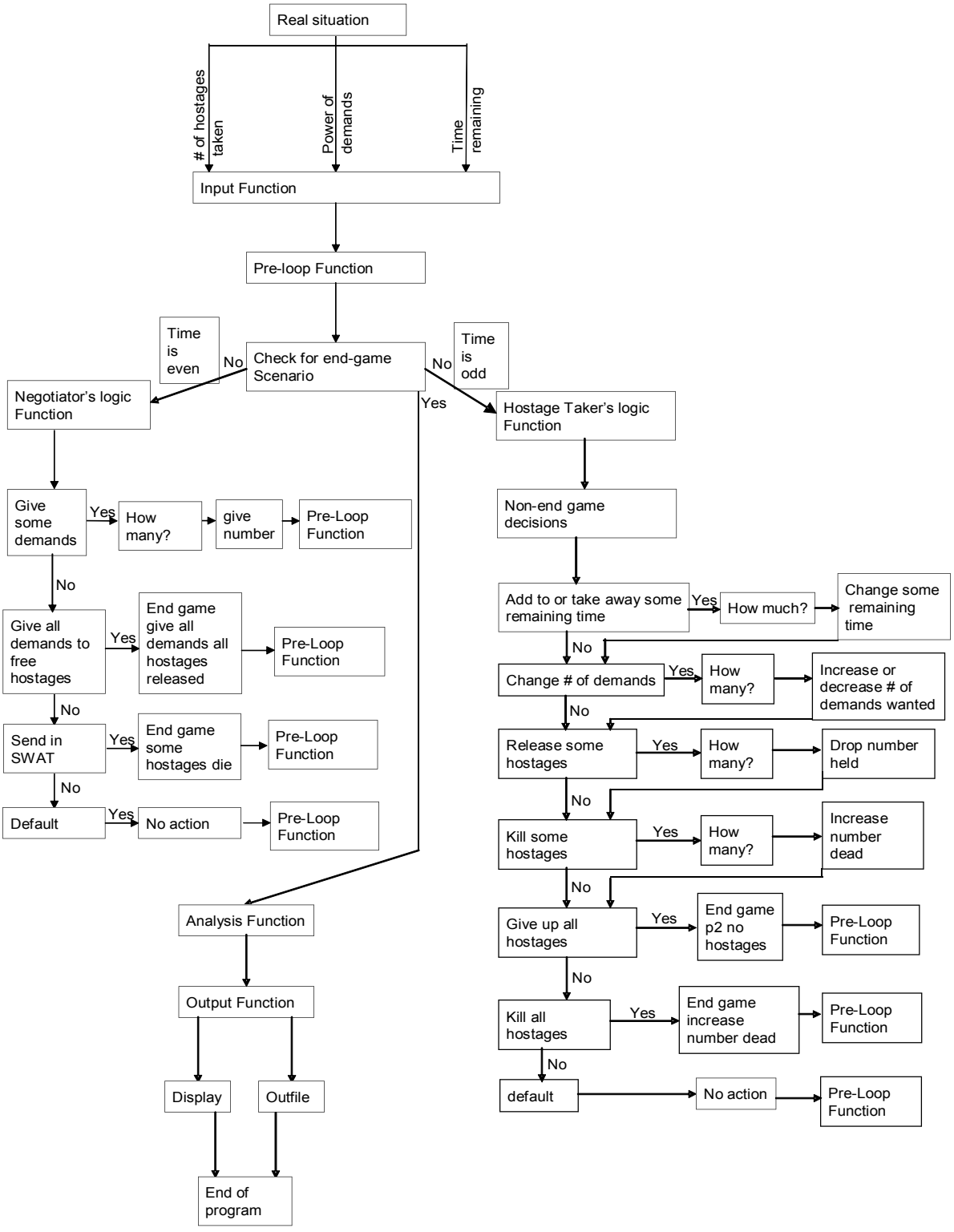
trms= terms

$S^{+}A$= Severity of positive actions

$S^{-}A$= Severity of negative actions

SConst= Severity Constant

Appendix 2: Flow chart of program 1



Appendix 3: Source Code for First program

```
// Team 60                    Source Code                    2/16/07

/* hostage2_0.cpp is the second attempt of team 60 at a program to model a
hostage crisis.  It is to demonstrate the capacity of variable probability and
weighted values, along with displaying probability changes.
*/

#include <iostream>
#include <cmath>
#include <string>
#include <iomanip>
#include <stdlib.h>
#include <time.h>
#include <fstream>
#include <math.h>

using namespace std;

ofstream output;

// #define FILE_IN "HostageData_Fin.in"
#define FILE_IN "HostageData_101.txt"

/****************************** Decisions_Class ******************************/

void wait_ms(int wait_time)
{
 clock_t start_time = clock() / (CLOCKS_PER_SEC / 1000);
 while ((clock() / (CLOCKS_PER_SEC / 1000)) - start_time < wait_time)
 {
      asm( "nop" );
 }
 return;
}

int fib(int num)
{
   if (num < 0) return -1;
   if (num == 0) return 0;
   if (num == 1) return 1;
   return fib(num - 1) + fib(num - 2);
}
class Decisions_Class
{
```

```
int h, d, tl;              // h = initial number of hostages
                           // d = initial number of demands
                           // tl = initial time left
int s1, s2, s3;            // s1 = switch var for Preloop_funct
                           // s2 = switch var for NegLog_funct
                           // s3 = switch var for HTakLog_funct
int lrand, hirand;         // lrand = low var for RandomNumber_f
                           // hirand = high var for RandomNumber_f
float p1[100], p2[100], p3[100], p4[100], p5[100], p6[100], p7[100], p8[100],
    p9[100];
                           // p1 = probvar for NegDemand_funct
                           // p2 = probvar for NegGiveIn_funct
                           // p3 = probvar for NegShootUp_funct
                           // p4 = probvar for HTakerTimeChange_f
                           // p5 = probvar for HTakerDemandChange_f
                           // p6 = probvar for HTakerHPartRelease_f
                           // p7 = probvar for HTakerHPartKill_f
                           // p8 = probvar for HTakerGiveUpH_funct
                           // p9 = probvar for HTakerHSlaughter_f

public:
int tpassed, ttaken, hdead[100], dwanted[100], hP2[100], dP2[100], tleft[100];
                           // tpassed = time unit gone by
                           // tpassed = PreLoop; for loop var
                           // ttaken = time of end
                           // hdead = # of h's killed
                           // dwanted = power of demands wanted
                           // hP2 = # of h's in P2's posession
                           // dP2 = power of demands given to P2
                           // tleft = time before taker shoots all h's
int hfreed[100], ddropped[100];
                           // hfreed = # hostages freed
                           // ddropped = demands that the taker dropped
float failrate;

void Input_funct(void);

void PreLoop_funct(void);

void NegotiatorLogic_funct(void);     // Negotiator = player1 = P1
void HTakerLogic_funct(void);         // HTaker = player2 = P2

int RandomNumber_funct(int lrand, int hirand);
// int WieghtedRand_fucnt(int, int)   // in future
// Equation_funct()                   // to be decided (it's its own project!)
```

```
  void CurrentVar_funct(void);

  // NegProbChange_funct              // these will replace p1&p2 "Prob_f" s
  // HTakerProbChange_funct             // some vars may need to be passed

  void NegProb_funct(void);            // may be only temporary
  void HTakerProb_funct(void);         // may be only temporary

  void NegDemand_funct(void);
  void NegGiveIn_funct(void);
  void NegShootUp_funct(void);

  void HTakerTimeChange_funct(void);
  void HTakerDemandChange_funct(void);
  void HTakerHPartRelease_funct(void);
  void HTakerHPartKill_funct(void);

  void HTakerGiveUpH_funct(void);
  void HTakerHSlaughter_funct(void);

  void Analysis_funct(void);
  void Output_funct(void);
};

/********************************* int main ***********************************/

int main()
{
  Decisions_Class Decisions_Class1;

  Decisions_Class1.Input_funct();
  Decisions_Class1.PreLoop_funct();
  Decisions_Class1.Analysis_funct();
  Decisions_Class1.Output_funct();

  system("PAUSE");

  return 0;
}

/******************************** Input_funct *********************************/

void Decisions_Class::Input_funct(void)
{
  cout<<"Input number of hostages taken: "<<endl;
  // cin >> h;
```

```
  h = 10;                          // soon we need to test with the inputs
  hP2[0] = h;

  cout<<"Input power of demand (1 to 10): "<<endl;
  // cin >> d;
  d = 10;                          // soon we need to test with the inputs
  dwanted[0] = d;

  cout<<"Input time remaining before finalization: "<<endl;
  // cin >> tl;
  tl = 10;                         // soon we need to test with the inputs
  tleft[0] = tl;

/* this may be where we start to input variabilities in the probabilities
based on the MENTAL STABILITY P2, TYPE OF CRISIS, etc.
    Mental stability P2;
       - clinically insane
       - under stress
       - cool
    Type of Crisis;
       - Terrorist
         - show-of-power  ex. taking only to decapitate/kill etc
         - Prisoners    ex. taking hostages to get the release of prisoners
         - suicide    ex. 9/11/2001
       - governmental  (? real type ?) (not ness in this program)
         - POW    ex. taking hostages to get the release of POWs
         - peace  ex. forcing peace by the release of POWs
       - civil
         - individual escape  ex. holding hostages so that P2 can escape
         - sexual exploitation  ex. taking for molestation(etc)
         -
    (other variables and types are possible)
Of course that would require a set up of string inputs or numerical value
switch for each type of situation.
*/

  dP2[0] = 0;
  hdead[0] = 0;
}

/******************************** PreLoop_funct ******************************/

void Decisions_Class::PreLoop_funct(void)
{
  s1 = 0;
  for (tpassed = 1; tpassed <= 100; tpassed++) {
```

```
    CurrentVar_funct();
    tleft[tpassed] = tleft[tpassed - 1] - 1;

    if (tleft[tpassed] == 0) {
      hdead[tpassed] = hdead[tpassed] + hP2[tpassed];
      hP2[tpassed] = 0;
      s1 = 2;
    }

    switch( s1 ) {
      case 0 :
        NegotiatorLogic_funct();
        s1 = 1;
        break;

      case 1 :
        HTakerLogic_funct();
        s1 = 0;
        break;

      default :
        ttaken = tpassed;
        tpassed = 101;
    }

    if (tpassed == 100) {
      ttaken = tpassed;
    }

    if ((hP2[tpassed] == 0) && (tpassed != 101)) {
      ttaken = tpassed;
      tpassed = 100;
    }

    if (dwanted[tpassed] == 0) {
      ttaken = tpassed;
      tpassed = 100;
    }
  }
}

/*************************** NegotiatorLogic_funct ***************************/

void Decisions_Class::NegotiatorLogic_funct(void)
{
  NegProb_funct();
```

```
  s2 = 0;
  while (s1 == 0) {
    switch (s2) {

// 0 endturn; decide if to give demands then how many.

      case 0:
        NegDemand_funct();
        break;

// 1 endgame; give in to all demands (all hostages returned)

      case 1:
        NegGiveIn_funct();
        break;

// 2 endgame; go in and shoot up (random # of people die, Hl-randdead=Hsurvived)

      case 2:
        NegShootUp_funct();
        break;

// defualt; end turn

      default:
        s1 = 1;
// in future; #0 will be weighted proportional in the number of demands given
    }
  }
}

/******************************** HTakerLogic_funct ************************/

void Decisions_Class::HTakerLogic_funct(void)
{
  HTakerProb_funct();
  s3 = 0;
  while (s1 == 1) {
       //cout << "s3: " << s3 << endl;   // test...
    switch (s3) {

// 0 all functions of non-endgame capable of being done together

      case 0:
  // change time left (tleft <=20 always)
        HTakerTimeChange_funct();
```

```
  //  demands (temporary lower; future raise & lower)
       HTakerDemandChange_funct();

  // give hostages; similar process to one that will be used in changing Tleft
       HTakerHPartRelease_funct();

  // kill hostages.
       HTakerHPartKill_funct();

// 1 endgame; give up all hastages.

      case 1:
        HTakerGiveUpH_funct();
        break;

// 2 endgame; kill all hostages.

      case 2:
        HTakerHSlaughter_funct();
        break;

// default; end turn

      default:
        s1 = 0;
    }
  }
}

/************************** RandomNumber_funct ****************************/

int Decisions_Class::RandomNumber_funct(int lrand, int hirand)
{
  wait_ms(738);
  srand(clock() + fib(clock() % 10));
  return rand() % (hirand - lrand + 1) + lrand;
}

/*
  notes on the idea of Equation_funct();
The purpose of the possible Equation_funct() is to try and take equations and
output some numerical data.  This may require several types of Equation_funct()s
because there may be several types of equations needed to be used.  These would
be highly statistical in nature.
*/
```

```
/*************************** CurrentVar_funct ******************************/

void Decisions_Class::CurrentVar_funct(void)
{
    hdead[tpassed] = hdead[tpassed - 1];
    dwanted[tpassed] = dwanted[tpassed - 1];
    hP2[tpassed] = hP2[tpassed - 1];
    dP2[tpassed] = dP2[tpassed - 1];
    tleft[tpassed] = tleft[tpassed - 1];
    hfreed[tpassed] = hfreed[tpassed - 1];
    ddropped[tpassed] = ddropped[tpassed - 1];
}

/****************************** NegProb_funct ******************************/

void Decisions_Class::NegProb_funct(void)
{
  p1[tpassed] = .10;    // in future all probs will be variable
  p2[tpassed] = .001;
  p3[tpassed] = .001;
}

/***************************** HTakerProb_funct ******************************/

void Decisions_Class::HTakerProb_funct(void)
{
  p4[tpassed] = .55;    // in future all probs will be variable
  p5[tpassed] = .05;
  p6[tpassed] = .05;
  p7[tpassed] = .07;
  p8[tpassed] = .001;
  p9[tpassed] = .001;
}

/************************** 1 NegDemand_funct ******************************/

void Decisions_Class::NegDemand_funct(void)
{
  if (RandomNumber_funct(1, 1000)<= int(p1[tpassed] * 1000)) {
    int n1 = RandomNumber_funct(0, dwanted[tpassed]);
    dP2[tpassed] = dP2[tpassed] + n1;
    dwanted[tpassed] = dwanted[tpassed] - n1;
              // } future place of weighted prob for # given
    s1 = 1;
  }
```

```
  else {
    s2++;
  }
}

/************************** 2 NegGiveIn_funct ******************************/

void Decisions_Class::NegGiveIn_funct(void)
{
  if (RandomNumber_funct(1, 1000)<=int(p2[tpassed]*1000)) {
    dP2[tpassed] = dP2[tpassed] + dwanted[tpassed];
    hfreed[tpassed] = hP2[tpassed] + hfreed[tpassed];
    hP2[tpassed] = 0;

    s1 = 2;
  }

  else {
    s2++;
  }
}

/************************** 3 NegShootUp_funct ******************************/

void Decisions_Class::NegShootUp_funct(void)
{
  if (RandomNumber_funct(1, 1000)<=int(p3[tpassed]*1000)){
    int n3 = RandomNumber_funct(0,hP2[tpassed]);
    hfreed[tpassed]=hP2[tpassed]- n3 + hfreed[tpassed];
    hdead[tpassed] = hdead[tpassed] + n3;
    hP2[tpassed] = 0;

    s1 = 2;
  }

  else {
    s2++;
  }
}

/************************ 4 HTakerTimeChange_funct ***************************/

void Decisions_Class::HTakerTimeChange_funct(void)
{
  if (RandomNumber_funct(1, 1000)<=int(p4[tpassed]*1000)) {
```

```
   int n4 = RandomNumber_funct(0, 10);
            // future place of weighted prob for amount changed
   int sw4 = RandomNumber_funct(0, 1);
            // future place of weighted prob for amount changed
   int n4b;

   switch( sw4 ) {
     case 0 :
       n4b = n4;
       break;

     case 1 :
       n4b = (-1)*n4;
       break;

     default :
       cout << "error in HTakerTimeChange_funct!!" << endl;
   }

   if ((tleft[tpassed]+n4b)>20)  {     // tleft must be less than 20 u
     tleft[tpassed] = 20;
   }

   else if ((tleft[tpassed]+n4b)<=0)  {    // tleft must be greater than 0 u
     tleft[tpassed] = 0;
     hdead[tpassed] = hdead[tpassed] + hP2[tpassed];
     hP2[tpassed] = 0;
     s1 = 2;
   }

   else  {
     tleft[tpassed] = int(tleft[tpassed] + n4b);
   }
  }

  else {
  }
}

/************************ 5 HTakerDemandChange_funct ************************/

void Decisions_Class::HTakerDemandChange_funct(void)
{
  if (RandomNumber_funct(1, 1000)<=int(p5[tpassed]*1000)) {
    int n5 = RandomNumber_funct(0, dwanted[tpassed]);
    ddropped[tpassed]=ddropped[tpassed] + n5;
```

```
    dwanted[tpassed]=dwanted[tpassed] - n5;
    hfreed[tpassed] = hfreed[tpassed] + hP2[tpassed];
    hP2[tpassed] = 0;
   }     // temporarilly only lowering of the demands

   else {

   }
}

/************************ 6 HTakerHPartRelease_funct ************************/

void Decisions_Class::HTakerHPartRelease_funct(void)
{
  if (RandomNumber_funct(1, 1000)<=int(p6[tpassed]*1000)) {
    int n6 = RandomNumber_funct(0, hP2[tpassed]);
    hfreed[tpassed] = hfreed[tpassed] + n6;
    hP2[tpassed] = hP2[tpassed] - n6;
  }     // future place of weighted prob for number released

  else {

  }
}

/************************ 7 HTakerHPartKill_funct ***************************/

void Decisions_Class::HTakerHPartKill_funct(void)
{
  if (RandomNumber_funct(1, 1000)<=int(p7[tpassed]*1000)) {
    int n7 = RandomNumber_funct(0, hP2[tpassed]);
    hP2[tpassed] = hP2[tpassed] - n7;
    hdead[tpassed] = hdead[tpassed] + n7;
  }     // future place of weighted prob for number dead

  else {

  }
}

/************************ 8 HTakerGiveUpH_funct *****************************/

void Decisions_Class::HTakerGiveUpH_funct(void)
{
  if (RandomNumber_funct(1, 1000)<=int(p8[tpassed]*1000)) {
    hfreed[tpassed] = hP2[tpassed] + hfreed[tpassed];
```

```
   hP2[tpassed] = 0;
   dwanted[tpassed] = 0;

   s1 = 2;
  }

  else {
   s3++;
  }
}

/************************ 9 HTakerHSlaughter_funct ***************************/

void Decisions_Class::HTakerHSlaughter_funct(void)
{
  if (RandomNumber_funct(1, 1000)<=int(p9[tpassed]*1000)) {
   hdead[tpassed] = hP2[tpassed] + hdead[tpassed];
   hP2[tpassed]=0;

   s1 = 2;
  }

  else {
   s3++;
  }
}

/******************************** Analysis_funct ********************************/

void Decisions_Class::Analysis_funct(void)
{
  failrate = float(hdead[ttaken]/h*5)
   + float(dP2[ttaken]/(ddropped[ttaken]+dP2[ttaken]+dwanted[ttaken])*5);
}       // temporary formula

/******************************** Output_funct ********************************/

void Decisions_Class::Output_funct(void)
{
   cout << setw(24) << "P1" << setw(24) << "P2" << endl
      << setw(12) << "tpassed"  << setw(12) << "hdead" << setw(12) << "dwanted"
      << setw(12) << "hP2"  << setw(12) << "dP2" << setw(12) << "tleft"<< endl;

   output.open(FILE_IN, ios::out);
   output << setw(24) << "P1" << setw(24) << "P2" << endl
      << setw(12) << "tpassed"  << setw(12) << "hdead" << setw(12) << "dwanted"
```

```
      << setw(12) << "hP2"  << setw(12) << "dP2" << setw(12) << "tleft"<< endl;

  for (tpassed = 0; tpassed <= ttaken; tpassed++) {
    cout << setw(12) << tpassed  << setw(12) << hdead[tpassed]
         << setw(12) << dwanted[tpassed] << setw(12) << hP2[tpassed]
         << setw(12) << dP2[tpassed] << setw(12) << tleft[tpassed] << endl;

    output << setw(12) << tpassed  << setw(12) << hdead[tpassed]
           << setw(12) << dwanted[tpassed] << setw(12) << hP2[tpassed]
           << setw(12) << dP2[tpassed] << setw(12) << tleft[tpassed] << endl;
  }

  output.close();
  cout<<"rate of failure: "<<failrate<<endl;
}
```

Appendix 4: Program One Data Examples

```
                    P1                        P2
     tpassed       hdead     dwanted          hP2          dP2       tleft
           0           0          10           10            0          10
           1           0          10           10            0           9
           2           0           0            0            0          14
rate of failure: 0
```

```
                    P1                        P2
     tpassed       hdead     dwanted          hP2          dP2       tleft
           0           0          10           10            0          10
           1           0          10           10            0           9
           2           0          10           10            0          14
           3           0          10           10            0          13
           4           0          10            3            0          12
           5           0          10            3            0          11
           6           2          10            1            0           6
           7           2           8            1            2           5
           8           2           8            1            2           4
           9           2           8            1            2           3
          10           2           8            1            2           8
          11           2           8            1            2           7
          12           3           8            0            2           6
rate of failure: 0
```

```
                    P1                        P2
     tpassed       hdead     dwanted          hP2          dP2       tleft
           0           0          10           10            0          10
           1           0          10           10            0           9
           2           0          10           10            0          14
           3           0          10           10            0          13
           4           0          10           10            0          15
           5           0           2           10            8          14
           6           0           2            9            8          11
           7           0           2            9            8          10
           8           0           2            1            8           9
           9           0           2            1            8           8
          10           0           2            1            8          15
          11           0           2            1            8          14
          12           0           2            1            8          13
          13           0           2            1            8          12
          14           0           2            0            8          15
rate of failure: 0
```

```
                    P1                        P2
     tpassed       hdead     dwanted          hP2          dP2       tleft
           0           0          10           10            0          10
           1           0          10           10            0           9
           2           0          10           10            0          14
           3           0          10           10            0          13
           4           0          10           10            0          12
           5           0          10           10            0          11
           6           0           6            0            0           7
rate of failure: 0
```

- note: on the second and third examples the rates of failure should be higher because people were killed and demands were given.

Appendix 5: Source Code for Second Program

```
// Team 60                    Hostage_II.cpp                    4/21/2007
// Abandon all hope ye who enter

#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>

using namespace std;

ofstream output;

#define FILE_IN "HostageIIData.txt"

void Input_funct(void);
void Process_funct(void);
int ActionVal_funct(int player, int action, int TorR);

float mainarray[5][9][11];
/*
  RsubN = mainarray[1][][]          // Negotiator step relationship val
  RsubT = mainarray[2][][]          // Taker step relationship val
  Tension = mainarray[3][][]        // Step tension val
  sigma = mainarray[4][][]          // Severity for next step
*/

int main()
{
  Input_funct();
  Process_funct();

  system("PAUSE");

  return 0;
}

void Input_funct(void)
{
  int HnumX, HnumY, HnumZ;
  int HCasNumX, HCasNumY, HCasNumZ;
  int HDedNumX, HDedNumY, HDedNumZ;
  float PArmFrac, PUnArmFrac;
```

```
int PStrenMax;
int TakerNum, PersArmVal, WeaponPow;
int Dtype[21], Dsigma[21];
int n = 0;

int VH, VCas, VDed, DxDV;
float Perim;

// cout << "Mode of opperation (1=action tree(default), 2=user action input): "

// auto setup is presently modeled after Munich, though it misses a demand.

cout << endl << endl << "INPUTS (initial situation)" << endl << endl
    << "Civilian" << endl << "  number of children taken hostages: ";
cin >> HnumX;
//HnumX = 0;
cout << endl << "  number of adults taken: ";
cin >> HnumY;
//HnumY = 9;
cout << endl << "  number of elderly taken: ";
cin >> HnumZ;
//HnumZ = 0;

cout << endl << endl << "  number of children casualties: ";
cin >> HCasNumX;
//HCasNumX = 0;
cout << endl << "  number of adult casualties: ";
cin >> HCasNumY;
//HCasNumY = 1;
cout << endl << "  number of elderly casualties: ";
cin >> HCasNumZ;
//HCasNumZ = 0;

cout << endl << endl << "  number of children dead: ";
cin >> HDedNumX;
//HDedNumX = 0;
cout << endl << "  number of adults dead: ";
cin >> HDedNumY;
//HDedNumY = 2;
cout << endl << "  number of elderly dead: ";
cin >> HDedNumZ;
//HDedNumZ = 0;

cout << endl << endl << "Perimiter" << endl
    << "  fraction of perimiter armed (0-1): ";
cin >> PArmFrac;
```

```
//PArmFrac = .5;
PUnArmFrac = 1-PArmFrac;
if (PUnArmFrac == 0)
  {
   PUnArmFrac = 1;
  }
cout << endl << "  maximum strength of perimeter (1-10): ";
cin >> PStrenMax;
//PStrenMax = 3;

cout << endl << endl << "Taker" << endl << "  number of takers: ";
cin >> TakerNum;
//TakerNum = 8;
cout << endl << "  personal armament value (1-10): ";
cin >> PersArmVal;
//PersArmVal = 7;
cout << endl << "  weapon strength (1-10): ";
cin >> WeaponPow;
//WeaponPow = 1;

int diloop = 1;
string dianswer;
cout << endl << endl << "Demands";
cout << endl << "Demand Scale List (estimate severities of demands on scale "
          << "of 1-10):"
<< endl << "  1:  Basic demands; Food, water, cigarettes"
<< endl << "       Scale is based on proportionality to total number of hostages"
       << " and takers"
<< endl << "         ex. 100 hostages and takers, 20 need object = 2"
<< endl << "  2:  Complex basic wants; Medical Supplies, drugs"
<< endl << "       Scale is based on proportionality to total number of hostages"
       << " and takers"
<< endl << "         ex. 50 hostages and takers, 20 need object = 4"
<< endl << "  3:  Demand of law enforcement removal action; "
<< endl << "       Scale; 1. "Get the police out of the way." "
<< endl << "         - 10. "OUT NOW OR I'LL SHOOT!""
<< endl << "  4:  Money"
<< endl << "       Scale; use powers of ten; 1. 10 dollars  - 10. ten billion"
<< endl << "  5:  Medium level abnormal demands; ex. "free all squirrels from"
       << " the zoo!" "
<< endl << "       Scale; 1. from Albuquerque Zoo - 10. from all zoos in America"
<< endl << "  6:  Higher difficulty abnormal demands; ex. "shut down the "
       << "abortion" << endl << "      clinics.""
<< endl << "       Scale; extremity of demand, ex. 1. An abortion clinic in "
       << "McCoy " << endl << "         - 10. all clinics in America."
<< endl << "  7:  Sexual exploitation, etc; "
```

```
<< endl << "       Scale based on the number of hostages per taker"
<< endl << "  8:  Transportation with safe passage."
<< endl << "       Scale; 1. car  - 5. bus   - 10. Boeing 747 or cruise ship"
<< endl << "  9:  National Independence or ethic betterment"
<< endl << "       Scale; 1. better treatment  - 10. Form independent state"
<< endl << "  10: Psychotic Demands; Nuclear material or weapons, confirmed "
      << "assassination" << endl << "      of a major political figure etc."
<< endl << "       Scale; place 10 for all." << endl;

for (diloop = 1; diloop <= 20; diloop++)
{
  cout << endl << "  level (type) (1-10): ";
  cin >> Dtype[diloop];
  //Dtype[diloop] = 6;
  cout << endl << "  severity (1-10): ";
  cin >> Dsigma[diloop];
  //Dsigma[diloop] = 5;

  cout << endl << "  are there any other demands (yes, no)? ";
  //dianswer = "no";
  cin >> dianswer;
  if (dianswer == "no")
  {
    n = diloop;
    diloop = 21;
  }
}
cout << endl;

for (int i = 0; i < n; i++)    // summates all values of demand times severity
{
  DxDV += (Dtype[i] * Dsigma[i]);
}

VH = 5*HnumX + 2*HnumY + 4*HnumZ;            // value of the hostages total
VCas = 5*HCasNumX + 2*HCasNumY + 4*HCasNumZ;  // value of casualties total
VDed = 5*HDedNumX + 4*HDedNumY + 2*HDedNumZ;  // value of init dead total

Perim = (PStrenMax) * (PArmFrac) / (PUnArmFrac) + TakerNum*WeaponPow;
                                        // initial equations
mainarray[1][0][0] = 0;                          // RsubN
mainarray[2][0][0] = 0;                          // RsubT
mainarray[3][0][0] = (VH/DxDV + (VCas + VDed)/Perim);  // Tension
mainarray[4][0][0] = ( mainarray[3][0][0] / 4 );       // Sigma
}
```

```
void Process_funct(void)
{
 int turn;
 int TA1, NA1;
 int NsubAv[3][2], TsubAv[3][2];
 int TsubAvsum = 0, NsubAvsum = 0;
 int Tenpossum = 0, Tennegsum = 0;
 float Spossum[11], Snegsum[11];
 int constant;

 if(mainarray[4][0][0] < 0)
   {
    Snegsum[0] = mainarray[4][0][0];
    Spossum[0] = 0;
   }
 else
   {
    Spossum[0] = mainarray[4][0][0];
    Snegsum[0] = 0;
   }

 for(NA1 = 1; NA1 <= 8; NA1++)                    // Turn 1
 {
  TsubAvsum = 0, NsubAvsum = 0;
  Tenpossum = 0, Tennegsum = 0;
  turn = 1;

  if(turn % 2 == 1)
  {
   NsubAv[turn][0] = ActionVal_funct(0, NA1, 0);
   NsubAv[turn][1] = ActionVal_funct(0, NA1, 1);
   TsubAv[turn][0] = 0;
   TsubAv[turn][1] = 0;
  }
  else
  {
   TsubAv[turn][0] = ActionVal_funct(1, TA1, 0);
   TsubAv[turn][1] = ActionVal_funct(1, TA1, 1);
   NsubAv[turn][0] = 0;
   NsubAv[turn][1] = 0;
  }

  for(int i = 1; i <= turn; i++)
  {
   TsubAvsum = TsubAvsum + TsubAv[turn][1];
   NsubAvsum = NsubAvsum + NsubAv[turn][1];
```

```
  if(NsubAv[i][0] < 0)
  {
   Tennegsum = Tennegsum + NsubAv[i][0];
  }
  else
  {
     Tenpossum = Tenpossum + NsubAv[i][0];
  }

  if(TsubAv[i][0] < 0)
  {
   Tennegsum = Tennegsum + TsubAv[i][0];
  }
  else
  {
   Tenpossum = Tenpossum + TsubAv[i][0];
  }
}

if(Tennegsum == 0)
{
 Tennegsum = 1;
}

if(Snegsum[0] == 0)
{
 Snegsum[0] = 1;
}

mainarray[1][NA1][0] = TsubAvsum / 2 + NsubAvsum / turn;

mainarray[2][NA1][0] = NsubAv[turn][1] +((NsubAvsum - NsubAv[turn][1]) +
                mainarray[2][0][0]) / 2;

mainarray[3][NA1][0] = float(Tenpossum / fabs(Tennegsum) * (Spossum[0] /
                Snegsum[0]) + (mainarray[2][NA1][0] +
                mainarray[1][NA1][0]) * 2.5);

mainarray[4][NA1][0] = fabs((( fabs(mainarray[3][NA1][0]) -
                (mainarray[1][NA1][0] + mainarray[2][NA1][0]) / 2 )
                + mainarray[4][0][0]) / 4);

if(NA1 == 8)
{
 TA1 = 11;
```

```
}
else
{
 for(TA1 = 1; TA1 <= 10; TA1++)                    // Turn 2
 {
  TsubAvsum = 0, NsubAvsum = 0;
  Tenpossum = 0, Tennegsum = 0;
  turn = 2;

  if(mainarray[4][NA1][0] < 0)
  {
   Snegsum[TA1] = Snegsum[0] + mainarray[4][0][0];
   Spossum[TA1] = Spossum[0];
  }
  else
  {
   Spossum[TA1] = Spossum[0] + mainarray[4][0][0];
   Snegsum[TA1] = Snegsum[0];
  }
  //cout << "snegsum[TA1] " << Snegsum[TA1] << endl;
  //cout << "spossum[TA1] " << Spossum[TA1] << endl;

  if(turn % 2 == 1)
  {
   //cout << "1" << endl;
   NsubAv[turn][0] = ActionVal_funct(0, NA1, 0);
   NsubAv[turn][1] = ActionVal_funct(0, NA1, 1);
   TsubAv[turn][0] = 0;
   TsubAv[turn][1] = 0;
  }
  else
  {
   //cout << "2" << endl;
   TsubAv[turn][0] = ActionVal_funct(1, TA1, 0);
   TsubAv[turn][1] = ActionVal_funct(1, TA1, 1);
   NsubAv[turn][0] = 0;
   NsubAv[turn][1] = 0;
  }
  for(int i = 1; i <= turn; i++)
  {
   TsubAvsum = TsubAvsum + TsubAv[turn][1];
   NsubAvsum = NsubAvsum + NsubAv[turn][1];

   if(NsubAv[i][0] < 0)
   {
    Tennegsum += NsubAv[i][0];
```

```
      }
      else
      {
       Tenpossum += NsubAv[i][0];
      }

      if(TsubAv[i][0] < 0)
      {
       Tennegsum += TsubAv[i][0];
      }
      else
      {
       Tenpossum += TsubAv[i][0];
      }
     }

     if(Tennegsum == 0)
     {
      Tennegsum = 1;
     }

     if(Snegsum[TA1] == 0)
     {
      Snegsum[TA1] = 1;
     }

     mainarray[1][NA1][TA1] = TsubAvsum / 2 + NsubAvsum / turn;

     //cout << NsubAv[turn][1] << "   " << NsubAvsum << "   " << endl;
     mainarray[2][NA1][TA1] = NsubAv[turn][1] +((NsubAvsum - NsubAv[turn][1]) +
                        mainarray[2][0][0]) / 2;

  // cout << Tenpossum << "   " << Tennegsum << "   " << Spossum[TA1] << "   "
    //     << Snegsum[TA1] << endl;
     mainarray[3][NA1][TA1] = float(Tenpossum / fabs(Tennegsum) *
                        (Spossum[TA1]/ Snegsum[TA1]) +
                        (mainarray[2][NA1][TA1] +mainarray[1][NA1][TA1])
                        * 2.5);

     mainarray[4][NA1][TA1] = fabs((( fabs(mainarray[3][NA1][TA1]) -
                       (mainarray[1][NA1][TA1] + mainarray[2][NA1][TA1])
                       / 2) + mainarray[4][NA1][0]) / 4);
    }
   }
}
```

```
string NA[9], TA[11];
NA[1] = "1. give demand";
NA[2] = "2. offer/option";
NA[3] = "3. negotiate";
NA[4] = "4. talk  ";
NA[5] = "5. ask   ";
NA[6] = "6. push  ";
NA[7] = "7. refuse";
NA[8] = "8. SWAT (EG)";
TA[1] = "1. negotiated EG";
TA[2] = "2. peace offering";
TA[3] = "3. remove demand";
TA[4] = "4. talk/hold    ";
TA[5] = "5. limit comms  ";
TA[6] = "6. threaten     ";
TA[7] = "7. add demand   ";
TA[8] = "8. violent action";
TA[9] = "9. kill hostage ";
TA[10] = "10. total homicide (EG)";

output.open(FILE_IN, ios::out);
output << "Step 0" << "\t\t\t\t\tRn" << "\tRt" << "\tTen" << endl << endl;
output << "initial" << "\t\t\t\t\t" << mainarray[1][0][0] << "\t"
      << mainarray[2][0][0] << "\t" << mainarray[3][0][0] << endl << endl;
output << "Step 1" << "\t\t\t\t\tRn" << "\tRt" << "\tTen" << endl << endl
      << "Negotiator" << endl << "action 1" << endl << endl;
for(int i = 1; i <= 8; i++)
{
 output << NA[i] << "\t\t\t\t" << mainarray[1][i][0]<< "\t"
       << mainarray[2][i][0] << "\t" << mainarray[3][i][0]<< endl;
}

output << endl << "Step 2" << "\t\t\t\t\tRn" << "\tRt" << "\tTen" << endl
      << endl << "Negotiator" << "\tTaker" << endl << "action 1"
      << "\taction 1" << endl << endl;

for(int i = 1; i <= 8; i++)
{
 for(int t = 1; t <= 10; t++)
 {
  if(i == 8)
  {
   t = 11;
  }
  else
  {
```

```
      output << NA[i] << "\t" << TA[t] << "\t" << mainarray[1][i][t] << "\t"
            << mainarray[2][i][t] << "\t" << mainarray[3][i][t] << endl;
     }
    }
    output << endl;
  }
  output.close();
}

int ActionVal_funct(int player, int action, int TorR)
{
  // function hold relationship and tension values for each action

  int AVal_Array[2][11][2];

  // Negotiator:
  // 1. give demand
  AVal_Array[0][1][0] = -15;      // T
  AVal_Array[0][1][1] = 4;        // R
  // 2. offer/option
  AVal_Array[0][2][0] = -12;      // T
  AVal_Array[0][2][1] = 3;        // R
  // 3. negotiate
  AVal_Array[0][3][0] = -9;       // T
  AVal_Array[0][3][1] = 2;        // R
  // 4. talk
  AVal_Array[0][4][0] = -6;       // T
  AVal_Array[0][4][1] = 1;        // R
  // 5. ask
  AVal_Array[0][5][0] = 6;        // T
  AVal_Array[0][5][1] = -1;       // R
  // 6. push
  AVal_Array[0][6][0] = 9;        // T
  AVal_Array[0][6][1] = -2;       // R
  // 7. refuse
  AVal_Array[0][7][0] = 12;       // T
  AVal_Array[0][7][1] = -3;       // R
  // 8. SWAT (EG)
  AVal_Array[0][8][0] = 50;       // T
  AVal_Array[0][8][1] = -15;      // R

  // Taker:
  // 1. negotiated EG
  AVal_Array[1][1][0] = -30;      // T
  AVal_Array[1][1][1] = 15;       // R
  // 2. peace offering
```

```
  AVal_Array[1][2][0] = -12;       // T
  AVal_Array[1][2][1] = 5;         // R
  // 3. remove demand
  AVal_Array[1][3][0] = -9;        // T
  AVal_Array[1][3][1] = 2;         // R
  // 4. talk/hold
  AVal_Array[1][4][0] = -6;        // T
  AVal_Array[1][4][1] = 1;         // R
  // 5. limit comms
  AVal_Array[1][5][0] = 12;        // T
  AVal_Array[1][5][1] = -1;        // R
  // 6. threaten
  AVal_Array[1][6][0] = 6;         // T
  AVal_Array[1][6][1] = -3;        // R
  // 7. add demand
  AVal_Array[1][7][0] = 9;         // T
  AVal_Array[1][7][1] = -2;        // R
  // 8. violent action
  AVal_Array[1][8][0] = 15;        // T
  AVal_Array[1][8][1] = -4;        // R
  // 9. kill hostage
  AVal_Array[1][9][0] = 30;        // T
  AVal_Array[1][9][1] = -6;        // R
  // 10. total homicide (EG)
  AVal_Array[1][10][0] = 75;       // T
  AVal_Array[1][10][1] = -12;      // R
  return AVal_Array[player][action][TorR];
}

/* Appendix A: Action values
Negotiator:            T       R
give demand            -15     4
offer/option           -12     3
negotiate              -9      2
talk                   -6      1
ask                    6       -1
push                   9       -2
refuse                 12      -3
SWAT (EG)              50      -15

Taker:                 T       R
Negotiated EG          -30     15
peace offering         -12     5
remove demand          -9      2
talk/hold              -6      1
limit comms            12      -1
```

threaten 6 -3
add demand 9 -2
violent action 15 -4
kill hostage 30 -6
total homicide (EG) 75 -12
*/

/* Appendix B: Symbols for the program
A = action
Cas = casualties
D = demands
Ded = dead
H = Hostage
I = initial
N = Negotiator
P = Perimeter
Pow = power
R = Relationship
S = sigma = severity
Ten = Tension
V = Value
W = Weapons

x = children
y = adults
z = elderly

n = next
c = current
p = previous
*/

/* Appendix C: Situation Inputs
--Munich--
hostages
adults- 9
casualties
adults- 1 (v=2)
dead
adults- 2 (v=4)
perimeter strength 3
takers- 8
Vweapons(personal)- 7
Demand1= 6 V=5
Demand2= 3 V=6
*/

```
/* Appendix D: Numbers per  step

  n       highest     sum          * 3           pages(est)
  0       1           1            3             1
  1       8           9            27            1
  2       70          79           237           2
  3       448         538          1614          11
  4       3920        4458         13374         90
  5       25088       29546        88638         591

  (7/8 * 10,  8/7 * 8)
*/

/* Appendix E: Early arrangement of the equations
  --General Variables--
// all with [] blank are arrays corresponding with steps

int TsubARv[2]                          //
int NsubARv[2]                          //
NsubARv[recent]                        //

  --General Equations--------> will go in Process_funct(void); -?

  mainarray[1][][] = (sum all TsubA[] R Vals)/18 + (sum all previous NsubA[] R vals)/
        (#terms of NsubA[] R vals);

  mainarray[2][][] = NsubA[n] + (sum NsubA[n-1 to 1])/2;

  mainarray[3][][] = (positive tension values) / fabs(negative tension values) *
           (sum (sigma+A) / sum (sigma-A) ) - (mainarray[2][][] + mainarray[1][][])/2 ;
// We really need to figure out how to sift between positive and negative vals

  mainarray[4][next][] = (( fabs(mainarray[3][current][]) - (mainarray[1][current][] +
mainarray[2][current][]) / 2 )
          + ( fabs(NegTenVal[n]) - NegRVal[n]) * (ActionVal[n]) + constant;
*/
```

Appendix 6: Program 2 Data Examples

Step 0		Rn	Rt	Ten
initial		0	0	0

Step 1		Rn	Rt	Ten
Negotiator action 1				
1. give demand		4	6	25
2. offer/option		3	4.5	18.75
3. negotiate		2	3	12.5
4. talk		1	1.5	6.25
5. ask		-1	-1.5	-6.25
6. push		-2	-3	-12.5
7. refuse		-3	-4.5	-18.75
8. SWAT (EG)		-15	-22.5	-93.75

Step 2		Rn	Rt	Ten
Negotiator action 1	Taker action 1			
1. give demand	1. negotiated EG	9	2	27.5
1. give demand	2. peace offering	4	2	15
1. give demand	3. negotiate	3	2	12.5
1. give demand	4. remove demand	3	2	12.5
1. give demand	5. talk/hold	2	2	10
1. give demand	6. limit comms	2	2	10
1. give demand	7. threaten	1	2	7.5
1. give demand	8. add demand	1	2	7.5
1. give demand	9. violent action	0	2	5
1. give demand	10. kill hostage	-1	2	2.5
1. give demand	11. total homicide (EG)	3	4.5	18.75
2. offer/option	1. negotiated EG	8	1.5	23.75
2. offer/option	2. peace offering	3	1.5	11.25
2. offer/option	3. negotiate	2	1.5	8.75
2. offer/option	4. remove demand	2	1.5	8.75
2. offer/option	5. talk/hold	1	1.5	6.25
2. offer/option	6. limit comms	1	1.5	6.25
2. offer/option	7. threaten	0	1.5	3.75
2. offer/option	8. add demand	0	1.5	3.75
2. offer/option	9. violent action	-1	1.5	1.25
2. offer/option	10. kill hostage	-2	1.5	-1.25
2. offer/option	11. total homicide (EG)	2	3	12.5
3. negotiate	1. negotiated EG	8	1	22.5
3. negotiate	2. peace offering	3	1	10
3. negotiate	3. negotiate	2	1	7.5
3. negotiate	4. remove demand	2	1	7.5
3. negotiate	5. talk/hold	1	1	5
3. negotiate	6. limit comms	1	1	5
3. negotiate	7. threaten	0	1	2.5
3. negotiate	8. add demand	0	1	2.5

3. negotiate	9. violent action	-1	1	0
3. negotiate	10. kill hostage	-2	1	-2.5
3. negotiate	11. total homicide (EG)	1	1.5	6.25

4. talk	1. negotiated EG	7	0.5	18.75
4. talk	2. peace offering	2	0.5	6.25
4. talk	3. negotiate	1	0.5	3.75
4. talk	4. remove demand	1	0.5	3.75
4. talk	5. talk/hold	0	0.5	1.25
4. talk	6. limit comms	0	0.5	1.25
4. talk	7. threaten	-1	0.5	-1.25
4. talk	8. add demand	-1	0.5	-1.25
4. talk	9. violent action	-2	0.5	-3.75
4. talk	10. kill hostage	-3	0.5	-6.25
4. talk	11. total homicide (EG)	-1	-1.5	-6.25

5. ask	1. negotiated EG	7	-0.5	16.25
5. ask	2. peace offering	2	-0.5	3.75
5. ask	3. negotiate	1	-0.5	1.25
5. ask	4. remove demand	1	-0.5	1.25
5. ask	5. talk/hold	0	-0.5	-1.25
5. ask	6. limit comms	0	-0.5	-1.25
5. ask	7. threaten	-1	-0.5	-3.75
5. ask	8. add demand	-1	-0.5	-3.75
5. ask	9. violent action	-2	-0.5	-6.25
5. ask	10. kill hostage	-3	-0.5	-8.75
5. ask	11. total homicide (EG)	-2	-3	-12.5

6. push	1. negotiated EG	6	-1	12.5
6. push	2. peace offering	1	-1	0
6. push	3. negotiate	0	-1	-2.5
6. push	4. remove demand	0	-1	-2.5
6. push	5. talk/hold	-1	-1	-5
6. push	6. limit comms	-1	-1	-5
6. push	7. threaten	-2	-1	-7.5
6. push	8. add demand	-2	-1	-7.5
6. push	9. violent action	-3	-1	-10
6. push	10. kill hostage	-4	-1	-12.5
6. push	11. total homicide (EG)	-3	-4.5	-18.75

7. refuse	1. negotiated EG	6	-1.5	11.25
7. refuse	2. peace offering	1	-1.5	-1.25
7. refuse	3. negotiate	0	-1.5	-3.75
7. refuse	4. remove demand	0	-1.5	-3.75
7. refuse	5. talk/hold	-1	-1.5	-6.25
7. refuse	6. limit comms	-1	-1.5	-6.25
7. refuse	7. threaten	-2	-1.5	-8.75
7. refuse	8. add demand	-2	-1.5	-8.75
7. refuse	9. violent action	-3	-1.5	-11.25
7. refuse	10. kill hostage	-4	-1.5	-13.75
7. refuse	11. total homicide (EG)	-15	-22.5	-93.75

Appendix 7: Bibliography

answers.com, “List of Hostage Crises.”

Bob Cordwell’s 2003 AiSC Project,
http://www.challenge.nm.org/archive/03-04/FinalReports/96.pdf

Grabianowski, Ed. people.howstuffworks.com, “How Hostage Negotiation works.”

Uschan, Michael V. Terrorism in Today’s World: the Beslan School Siege and Separatist Terrorism, World Almanac Library. Milwaukee, Wisconsin, 2006.

wikipedia.com