CAPsim Tutorial

Tom Baker

Technical Report CS/E-88-032
20 September 1988

Oregon Graduate Center
19600 S.W. von Neumann Drive
Beaverton, Oregon 97006-1999

CAPsim Tutorial

Tom Baker
Dept. of Computer Science & Engineering
Oregon Graduate Center
Beaverton, Oregon

1. Introduction

CAPsim is an artificial neural network (ANN) simulator library. Its main purpose is to study the effects of hardware implementation on neural network algorithms. It can also be used to create and simulate ANN applications. The CAPsim library consists of many useful building blocks for neural network simulation. CAPsim is not a single network simulator, nor is it a program that allows one to dynamically create networks and topologies.

This tutorial is written to describe CAPsim and to help users create and simulate their own ANN's. Many of the necessary network 'building blocks' have already been implemented. These 'building blocks' make construction of ANN applications much easier. Without a description of the design and functionality of the library much of the usefulness would be wasted.

Section 2 discusses some of the considerations in designing the CAPsim library. Section 3 describes some of the 'building blocks' that are available for building ANN's. Section 4 has a description of the user interface library of functions and how they can be used to define menu's and custom simulator functionality. Section's 5 and 6 show the user how to construct a custom simulator. Section 7 is a directory of the CAPsim library.

2. CAPsim Design Philosophy

The two strongest considerations in the design of CAPsim are code modularity and execution speed. The simulators that are implemented with the CAPsim library need to be easily constructed, so many simple modules that interface in a clean and flexible way should be used. Networks can be constructed with a few simple definition statements. The ANN simulators must also execute as fast as possible. ANN simulations on conventional hardware are typically slow and compute intensive. Most of the execution time for the simulators is spent in a few small computation loops. Although most of the CAPsim library is programmed for readability, the computation loops are programmed for speed.

The C++ programming language was chosen as an implementation language because of the combination of object oriented language features and efficient target code. The C++ compiler is a preprocessor that produces a C program. One basic data object of the C++ language is the class. A class is a data structure that has a state (variables) and an interface definition (member functions). The member functions define the operations that may be performed on the state variables.

This research supported by SRC grant 86-10-097.

The object oriented feature of inheritance makes the task of programming modular code much easier. Inheritance lets a class use the variables and member functions of another class. A class can be written that will define the interface between two software modules. All classes that are defined to inherit the member functions can use the same code to communicate. The communication code can be written once, so that the interface functions do not have to be rewritten for each new function that is programmed. The class that defines the code to be inherited is referred to as the superclass, and the class that inherits the code is known as the subclass.

The key to modularity in the CAPsim library is the common superclass vector. A vector represents one layer in an ANN network. The vector class is implemented as a one-dimensional array. Higher dimensional layers can be implemented easily with the vector interface. All ANN application classes inherit the vector class member functions. The values in the vector represent the activation values of the nodes in the network. Each application class reads the values of the layers that it is connected to.

A feature of C++ that helps produce fast programs is that of inline functions. During the compilation of a program, the code that would normally be executed in a procedure is inserted directly into the code. Most of the execution time for the simulators occurs in calculations in a few loops. Inline functions are used to keep function calls within these loops to a minimum. The use of inline functions helps make the code within the calculation loops readable, modular and fast.

The execution time for many of the simulations is about several hours, so the user interface has been designed for both interactive execution and batch processing. The emphasis for the user interface is to provide methods for setting key simulation variables and tracing intermediate values. The simulation variables are used to define the algorithm modifications that are to be simulated without having to recompile the code. While the simulation executes, trace values are printed periodically to show the performance of the ANN algorithm. The CAPsim library makes custom menus and user input functions easy to program so that the user interface can be tailored to the ANN application.

3. Network Building Blocks

3.1. vector, floatVector and integerVector

The class vector is a superclass to all simulation objects that implement ANN algorithms. Vector defines the interface to the values of the nodes in each layer of the network. The values are represented internally as a one dimensional array. All values that are sent between layers of an ANN application use the member functions that are defined in vector. The vector class is really only an interface specification. The only data that is contained in this class is vector size. The only member functions used in the vector class are for printing vector values.

The actual values of the vector are contained in either floatVector or integerVector. These classes contain the arrays of values for the vector. The class floatVector defines an array of type float. FloatVector is used for most general ANN applications. The class integerVector defines an array of type int. IntegerVector is used for the simulating a hardware architecture that uses only integer arithmetic. The values contained in integerVector are really integer representations of floating point values. The values carry eight bits of precision, all the bits are to the right of the decimal point. When a new class is created it should be defined as a subclass

of floatVector or integerVector, instead of being a subclass of vector.

The following member functions are the main interface for the classes floatVector and integerVector:

float GetValue(int) - GetValue() returns the float value of the element referenced by the passed parameter. This function is used in both floatVector and integerVector. In the class integerVector the value is converted to the floating point representation.

int IntegerValue(int) - IntegerValue() is only a member function of the class integerVector. This function returns the integer value referenced by the passed parameter.

void PutValue(int,float) - PutValue() stores the floating point value passed as the second parameter into the array element referenced in the first parameter. This function is used in both floatVector and integerVector. In the class integerVector, the value is converted into integer representation before it is stored in the array.

void PutValue(int,int) - This version of PutValue() is used only in the class integerVector. The second parameter is stored directly into the array without type conversion.

void WriteVector(FILE) - The function WriteVector() prints the values of the vector onto the stream passed as the parameter. The length of the vector is printed first, then the values. This function is defined in the class vector. All array elements are written as floating point values.

void WriteShortVector(FILE) - WriteShortVector() writes an abbreviated version of the vector onto the stream passed as the parameter. The base ten digit to the right of the decimal point is printed (tenths). This function is used when long vectors are displayed and precision is not desired. This function is defined in the class vector.

3.2. Constants, lib

The files 'constants.h', 'lib.h', and 'lib.c' contain global variables and functions that are used throughout the CAPsim library. Many variables and functions are not specific to any class, so they are placed in these library files. The file 'constants.h' has global variables and type definitions. The system I/O definition file 'stdio.h' is also included in 'constants.h'. The files 'lib.h' and 'lib.c' defines global functions that are used in the CAPsim library. Most of the functions are defined as inline functions to generate fast executable code.

3.3. Model Classes

A *Model Class* is a class that specifies the functionality and topology of a network. Each model class contains the data and member functions needed to interface with a specific ANN application. All user interaction of the network is conducted through the model class.

Among data items defined by the model class are pointers to all the layers of the network. In the class constructor, the network layers are allocated and linked together. Each layer class that reads values from another layer must have a member function that receives a pointer to a vector as a parameter. These member functions are the way that the layers of a network are connected together.

Another feature of the model class is a member function that executes one pass through the network. This member function is called Run(). There may be options that specify exactly what is executed by the function Run(). For example, a back-propagation may just run the feed forward operations or it may run feed forward and error propagation. One advantage to having a model class is that the user interface has to communicate with only one class to modify global network parameters.

3.4. I/O classes

I/O classes are for communication between the network and the hypothetical external environment. These classes are subclasses of the class vector. An I/O class presents input vectors to the network.

3.4.1. inclass

The inclass class reads a vector from a file and puts the values into its value array. Inclass keeps an input buffer for the input vectors. There are parameters that define the buffer size and the number of times to repeat the buffer before reading new data.

3.4.2. outclass

The outclass class writes the values of the input vector to a file. The outclass is connected to another vector class that it reads from. Outclass is for logging vector values to a file.

3.4.3. vectorgen

The class vectorgen is an abstract superclass that generates input and target vector pairs. Vectorgen is a subclass of floatVector and its values are the target vectors for the network. There is also a member variable of vectorgen that is a pointer to a floatVector. This member variable is called inputs. There is a member function that returns the pointer inputs so that another vector class can use the values that inputs points to. Vectorgen is used when the target and input values are calculated rather than read from a file.

3.5. Back-propagation Layers

There are several network layers that have been designed specifically for the back-propagation algorithm. These layers are subclasses of vector and can be linked with any of the previously mentioned classes to form a network. A typical back-propagation network is made up two forward class instances, an error class instance and a backward class instance. The forward class instances calculate the hidden layer and the output layer for the network. The error class calculates the delta value for the output layer, and the backward class calculates the delta values for the hidden layer. Additional hidden layers can be added by inserting more forward and backward instances. The inputs and targets to the network can be provided by two instances of inclass, or one instance of vectorgen. Figure X shows how a typical back-propagation network is connected. Network execution contains two phases. The first phase calculates the hidden and output values of the network. The second phase calculates the error values of the output and hidden layers, and computes the weight changes.

3.5.1. forward

The forward vector participates in both phases of execution. During the forward phase the activation values of the nodes in the layer are calculated. The value (o_i) for each node is $o_i = f(\sum_{j=0}^{n} o_j w_{ij})$, where n is the number of input nodes, o_j is the value of input j, and w_{ij} is the connection weight between input j and node i. The non-linear 'squishing' function is $f(x) = \frac{1}{1+e^{-x+\theta}}$ where θ is a learned threshold value. During the backward error phase, the forward vector computes the modifications to the weights. The weight changes are $\Delta w_{ij} = \eta \delta_i o_j + \alpha \Delta w_{ij}$, where η is the learning rate, δ is the error gradient (see error and backward classes), and α is the momentum rate.

3.5.2. error

The error class computes the error gradient for the output vector of the back-propagation network. The error gradient for each output node (o_i) is $\delta_i = \left(t_i - o_i\right) o_i \left(1 - o_i\right)$, where t_i is the desired value for the node.

3.5.3. backward

The backward class computes the error gradient for a hidden vector of the back-propagation network. The error gradient for the output of the node o_i is $\delta_i = o_i \left(1 - o_i\right) \sum_{j=1}^{m} \delta_j w_{ij}$, where δ_j is the error gradient the node that is connected by the weight w_{ij}.

4. User Interface

The user interface classes allow the programmer to build custom interfaces. The classes described here handle the I/O details of a menu driven interface. Special menus and menu selections are easily defined and implemented.

4.1. menuItem

The class menuItem represents one selection of an entry in a menu. There are four parameters given to the constructor of a menuItem: command, keyword, description, and menuAction. The first three parameters are pointers to strings, the fourth parameter is a pointer to a function. The command parameter is the input string that is used to select the menuItem. The keyword parameter is a short string that is used to identify the menuItem. This parameter is intended for use in pop-up menus. The description parameter is a long string that defines the use of the menuItem. The menuAction parameter points to the function to be called when the menu selection is made.

There are two important member functions that are defined by menuItem. They are action and help. The action() function is called when the menuItem has been selected. This function simply calls the function that the pointer menuAction references. The help() function prints out a string that defines the instance of menuItem. The first three parameter strings are printed, separated by tabs.

4.2. menu

The menu class manages a collection of menuItems. This class is implemented by using an array of pointers to menuItems. The only parameter that is passed to the constructor is the size of the array (the number of menuItems that can be referenced). The member functions for the menu class provide for adding new menuItems, selecting a particular menuItem in the list, performing the action of a selected menuItem and calling the help member function of each menuItem in the list.

4.3. keymenu

The class keymenu is used for receiving command strings from the standard input and performing the appropriate action. There is a pointer to a function that prints the prompt string for the user input. The prompt function is user defined and can be used to print the current state of the simulation (clocks, cycles, etc.). Keymenu contains a pointer to a menu that is used to execute the command strings that are input. The member function batch() reads in a string, parses the first word from the string and calls the menu action() function. The keymenu class can also read and execute Macro data structures.

4.4. window

The class window provides an object oriented interface with the Curses window library. The Curses library is a standardized terminal interface that provides the capability to create windows, and output data in user defined locations of a terminal screen. The data contained in the window class includes a window pointer, the size of the window and the origin of the window. Many of the functions of the Curses library are implemented in the window class.

5. Creating a Layer Class

There are several points that should be taken into consideration when creating an ANN layer class for CAPsim. An important decision is what superclass to use. All layer classes must use the vector class as a definition of inter-layer communication. The class floatVector is the most common superclass for initial versions of a layer class. Normally it is easier to program and debug an algorithm with floating point representations than integers. Once the ANN calculations have been tested, an equivalent layer can be written with the integerVector superclass if integer calculations are desired.

Next, the connections with other layers should be defined. The class should have a function that connects it with all other layers from which it is receiving data. The connection can be implemented by assigning a pointer to a layer that is a subclass of the vector class. Once a variable is assigned to point to another layer, the values of the layer can be read by indirectly calling the GetValue(), and IntegerValue() member functions. The member function that connects an input layer should check that the size of the layer is correct. The simulator will run much faster if bounds checking is done at the time that the network is connected, rather than when the network is performing its calculation functions.

The activation functions can be programmed after the layer connections have been defined. Often there is a single function that performs the calculations for a layer. Member functions that set options and modify computation variables may be useful, also.

example - feed forward class

```
/* feedforward

  This class performs a simple feed forward calculation.  It is
  a subclass of floatVector.

*/

class feedforward : public floatVector
{
  floatVector*   inputs;                  // pointer to input layer
  float* weights;          // connection strengths
  int              inputSize;             // expected size of input vector

public:

  feedforward::feedforward(int,int);
  void ConnectInput(floatVector*);        // connect input layer
  void CalculateVector();                 // compute activation values

};

feedforward::feedforward(int insz, int sz) : (sz)
{
  int i;

  inputSize = insz;
  weights = new float[inputSize * GetSize()];  // allocate weight array
  for (i = 0; i < inputSize * GetSize(); i++)
    weights[i] = 1.0;
}

void feedforward::ConnectInput(floatVector* inputPointer)
{

  if (inputSize != inputPointer -> GetSize())
  {
    printf(stderr,"feedforward::ConnectInput: input size mismatch0);
    exit(0);
  }

  inputs = inputPointer;
}

void CalculateVector()
{
  int i,j;
  float temp;
  float* base = weights;

  for (i = 0; i < GetSize; i++)                    // for each value
  {
```

```
    for (j = 0; j < inputSize; j++)                    // for each input
    {
      temp = inputs -> GetValue(j) * *base++;          // calculate value
    }
    PutValue(i,temp);                                  // store value into vector
  }
}
```

6. Creating a Program

Making an ANN simulation is an easy process once the layer classes have been programmed. All that is involved is to assemble the standard classes and the application specific code into a program unit. There are three parts of code that are needed for the program: the model class, the user interface, and the main program block. The model class allocates and executes the application specific classes. The user interface receives user input and calls functions to do the desired operations. The main program block allocates the data structures of the simulator and start the execution of the program.

6.1. Model

The model class contains the data structures for the ANN application. The data contained by the model class includes pointers to instances of the layer classes. The constructor for the model class creates new instances of the layer classes and links them together. One important member function of the model class is one that will execute a single pass of the network. The model class must also define member functions that will be used as an interface for the layer classes. Any member functions that a layer class has that will allow the user to change parameters must be accessible through the model class.

6.2. User Interface

The classes supplied with CAPsim that support the user interface are designed for the programmer to have flexibility in the functionality of the simulation. The programmer can provide as much or as little freedom as possible. The user interface will issue the run commands of the model class, and will also allow the user to modify parameters of the simulation.

All menu options are created by defining instances of the class menuItem. The parameters of the menuItem definition include the string that the user will enter to invoke the operation, and a pointer to the function that will be executed when the menuItem is selected. It is up to the programmer to supply the functions that are called by the instances of menuItem. There should also be a definition of one or more instances of the class menuItem. The instances of menuItem will be assigned to instances of menu in the main program block. An instance of keyMenu is the only other part of the user interfaces that is required. An instance of menu will be assigned to the keyMenu instance in the main program block.

6.3. Main

The main program block is mainly used for data structure definition and allocation. One important global data structure is a character array instring, which is used by the user interface for the input command string. The main program block should not contain much program code, most of the critical functions should be defined elsewhere. The only operations that the main block should do is to initialize the network and user interface, and to invoke a keyMenu

instance member function for execution of the simulator.

Example Model Class

```
class model
{
  int bottomSize, middelSize, topSize;
  inclass* bottom;
  feedforward* middle;
  feedforward* top;

public:

  model(int,int,int);
  void cycle();
  void BindStream(FILE* fp) { inclass -> BindStream(fp); }
}

model::model(int b, int m, int t)
{
  bottomSize = b;
  middleSize = m;
  topSize = t;

  // allocate layers

  bottom = new inclass(bottomSize,1);
  middle = new feedforward(bottomSize,middleSize);
  topSize = new feedforward(middleSize,topSize);

  // link network

  middle -> ConnectInput(bottom);
  top -> ConnectInput(middle);
}

model::Cycle()
// run one forward pass through network
{

  bottom -> GetVector();
  middle -> CalculateVector();
  top -> CalculateVector();

 }
```

User Interface Example

```
//
//  User interface functions
//

void Run()
// run feedforward network one cycle
{

  m.Cycle();
}

void OpenInputFile()
// get file name from input, open file and call model
{
  FILE* fd;
  char fileName[256],dummy[256];

  sscanf("%s %s",dummy,fileName);
  if ((fd = fopen(fileName,"r")) <= 0)
    printf(stderr," Error occurred opening file <%s>0,fileName);
  else
    m.BindStream(fd);

}

// define menu selection items

menuItem r("r","run","Run network one feed forward pass",Run);
menuItem f("f","file","Read input data from file",OpenInputFile);
```

Main block example

```
// Define user interface data structures

keyMenu keyMenuInstance();
menu menuInstance(5);

// Define network
int size1 = 5;
int size2 = 3;
int size3 = 2;

model modelInstance(size1,size2,size3);

main()
{
// initialize classes

  keyMenuInstance.setMenu(&menuInstance);
  menuInstance.addItem(&r);
  menuInstance.addItem(&f);

// run network

  keyMenuInstance.batch();

}
```

Appendix: CAPsim Library Class Directory

class alphabet : superclass vectorgen

	alphabet(int,int,int)
void	ReadFile()
int	GetVector()
void	WriteShortVector(FILE*)
void	WriteInputVector(FILE*)
void	asShortText(char*)

class backprop

	backprop(int,int,int)
void	setEta(float)
void	setAlpha(float)
int	clockValue()
int	vclkValue()
void	setRepeatCount(int)
void	setAccumulateCount(int)
void	GetInput()
void	setBufferSize(int)
void	setErrorTraceCount(int)
void	targetTrace(FILE*)
void	outTrace(FILE*)
void	hiddenTrace(FILE*)
void	errorTrace(FILE*)
void	inTrace(FILE*)
void	aveTrace(FILE*)
void	maxError()
void	ReadWeights(char*)
void	WriteWeights(char*)
void	SetRandomWeights(int)
void	SetRandomPositiveWeights(int)
void	SetRandomNegativeWeights(int)
void	Run()
void	runFullCycle()
void	runFull()
void	runFeedForward()
void	runForwardCycle()
void	backwardCycle()

class backward : superclass floatVector

	backward(int,int)
void	ConnectError(floatVector*)
void	ConnectOutput(floatVector*)
void	ConnectWeights(forward*)
int	UpdateValue()

class classify : superclass integerVector

	classify(int,int)
void	ConnectInput(prototype*)
void	ConnectTarget(integerVector*)

```
        void    UpdateValue()
        int     prototypeNumber(int)

class error : superclass floatVector

                error(int)
        void    ConnectOutput(floatVector*)
        void    ConnectTarget(floatVector*)
        void    RepeatCount(int)
        void    writeErrorTrace(FILE*)
        void    writeShortErrorTrace(FILE*)
        void    Clear()
        int     ComputeDelta()
        float   AverageDelta()
        float   ErrorSum()
        float   maxError()

class floatVector : superclass vector

                floatVector(int)
        float   GetValue(int)
        void    PutValue(int,float)

class fonts : superclass intVectorgen

                fonts(int,int,int)
        void    ReadFile()
        void    GetVector()
        void    WriteShortVector(FILE*)
        void    WriteInputVector(FILE*)
        void    asShortText(char*)

class forward : superclass floatVector

                forward(int,int)
        void    ConnectInput(floatVector*)
        void    ConnectOutput(forward*)
        void    ConnectError(floatVector*)
        floatVector* GetDelta()
        float*  WeightPointer()
        int     GetInsize()
        int     UpdateValue()
        int     UpdateWeights()
        void    ReadWeights(FILE*)
        void    ReadHiOrder(FILE*)
        void    WriteWeights(FILE*)
        void    WriteHiOrder(FILE*)
        void    initializeWeights()

class inclass : superclass floatVector

                inclass(int,int)
        void    BindStream(FILE*)
```

```
void    RepeatCount(int)
int     IsBound()
void    RepeatBuffer(int)
void    GetVector()
void    ReadVector()
```

class intAlphabet : superclass intVectorgen

```
        intAlphabet(int,int,int)
void    ReadFile()
int     GetVector()
void    WriteShortVector(FILE*)
void    WriteInputVector(FILE*)
void    GetAccumulatedInput()
void    GetAverageInput()
```

class intBackprop

```
        backprop(int,int,int)
void    setEta(float)
void    setAlpha(float)
int     clockValue()
int     vclkValue()
void    setRepeatCount(int)
void    setAccumulateCount(int)
void    GetInput()
void    setBufferSize(int)
void    setErrorTraceCount(int)
void    targetTrace(FILE*)
void    outTrace(FILE*)
void    hiddenTrace(FILE*)
void    errorTrace(FILE*)
void    inTrace(FILE*)
void    aveTrace(FILE*)
void    maxError()
void    ReadWeights(char*)
void    WriteWeights(char*)
void    SetRandomWeights(int)
void    SetRandomPositiveWeights(int)
void    SetRandomNegativeWeights(int)
void    Run()
void    runFullCycle()
void    runFull()
void    runFeedForward()
void    runForwardCycle()
void    backwardCycle()
void    SetErrorCount()
void    runAccumulate()
void    setWeightStepSize(float)
int     forwardWeightChanges()
void    useActualWeights()
void    useSignWeights()
void    useThresholdWeights()
```

```
        void    useAnnealWeights()

class intBackward : superclass integerVector

                intBackward(int,int)
        void    ConnectError(integerVector*)
        void    ConnectOutput(integerVector*)
        void    ConnectWeights(intForward*)
        int     UpdateValue()

class integerVector : superclass vector

                integerVector(int)
        float   GetValue(int)
        int     IntegerValue(int)
        void    PutValue(int,int)
        void    PutValue(int,float)
        int     SumValue(int)

class intError : superclass integerVector

                intError(int)
        void    ConnectOutput(integerVector*)
        void    ConnectTarget(integerVector*)
        void    RepeatCount(int)
        void    SetErrorCount(int)
        void    writeErrorTrace(FILE*)
        void    writeShortErrorTrace(FILE*)
        void    Clear()
        int     ComputeDelta()
        float   AverageDelta()
        float   ErrorSum()
        float   maxError()
        void    GetAccumulatedError()

class intForward : superclass integerVector

                intForward(int,int)
        void    ConnectInput(integerVector*)
        void    ConnectOutput(intForward*)
        void    ConnectError(integerVector*)
        vector* GetDelta()
        int*    WeightPointer()
        int     GetInsize()
        void    useAnnealWeights()
        void    useActualWeights()
        void    useSignWeights()
        void    useThresholdWeighs()
        void    dontPropagate()
        void    setWeightStepSize()
        void    setAnnealBits()
        void    setEta()
        void    setAlpha()
```

```
void    setAccumulateCount(int)
int     UpdateValue()
int     UpdateWeights()
int     updateAnnealWeights()
int     updateActualWeights()
int     updateSignWeights()
int     updateThresholdWeights()
int     AccululateWeights()
int     AccululatedUpdateWeights()
void    ReadWeights(FILE*)
void    ReadHiOrder(FILE*)
void    WriteWeights(FILE*)
void    WriteHiOrder(FILE*)
void    initializeWeights()
void    initializePositiveWeights()
void    initializeNegativeWeights()
```

class intOutclass : superclass integerVector

```
        intOutclass(int)
void    ConnectInput(integerVector*)
void    BindStream(FILE*)
void    UnBindStream(FILE*)
int     IsBound()
void    UpdateValue()
void    PutVector()
```

class intVectorgen : superclass integerVector

```
        intVectorgen(int,int,int)
void    RepeatCount(int)
integerVector* inputVector()
void    RepeatBuffer(int)
```

class keyMenu

```
        keyMenu()
        keyMenu(menu*)
void    setMenu(menu*)
void    batch()
void    setEcho()
void    setPrompt(void (*p)())
void    inputString()
void    readMacro(macro*)
void    executeMacro(macro*)
```

class macro

```
        macro(char*)
int     GetSize()
macro*  nextMacro()
void    link(macro*)
char*   inputString(int)
```

```
        char*   GetName()
        void    addCommand(char*)
        void    readFile()
        void    writeFile()

class menu

                menu(int)
        void    helpText(char*)
        int     GetSize()
        void    addItem(menuItem*)
        void    select(int)
        void    select(char*)
        void    help()
        void    action()
        void    action(int)
        void    action(char*)

class menuItem

                menuItem(char*,char*,char*,void (*p)())
        char*   key()
        char*   cmd()
        void    action()
        void    help()
        void    helpText()

class nestor

                nestor(int,int,int)
        void    targetTrace(FILE*)
        void    outTrace(FILE*)
        void    inTrace(FILE*)
        int     prototypes()
        void    Run()
        int     prototypeNumber(int)

class normal : superclass integerVector

                normal(int)
        void    ConnectInput(integerVector*)
        void    GetVector()

class outclass : superclass floatVector

                outclass(int)
        void    ConnectInput(integerVector*)
        void    BindStream(FILE*)
        void    UnBindStream(FILE*)
        int     IsBound()
        void    UpdateValue()
        void    PutVector()
```

class prototype : superclass integerVector

```
        prototype(int,int,float)
void    ConnectInput(integerVector*)
void    UpdateValue()
int     newPrototype(int)
void    confusion(int)
int     numberPrototypes()
int     max()
```

class vector

```
        vector(int)
int     GetSize()
float   GetValue()
void    WriteVector(FILE*)
void    WriteShortVector(FILE*)
char*   asText()
void    asShortText(char*)
```

class vectorgen : superclass floatVector

```
        vectorgen(int,int,int)
void    RepeatCount(int)
floatVector* inputVector()
void    RepeatBuffer(int)
```

class window

```
        window(int,int,int,int)
        ~window()
WINDOW*     windowPtr()
void    show()
void    addch(char)
void    addstr(char*)
void    box(char,char)
void    clear()
void    erase()
char    getch()
void    getstr(char*)
void    getxy(int*,int*)
void    move(int,int)
void    overwrite(window*)
void    refresh()
void    standend()
void    standout()
WINDOW*     subwin(int,int,int,int)
```

class windowMenu : superclass keyMenu

```
        windowMenu()
        windowMenu(menu*)
void    wind(window*)
```