The Hyper-Linear Package

Peter M. Maurer
Department of Computer Science
Baylor University
Waco, TX 76798

1 Abstract
The Hyper-Linear package is the core of the hyper-linear simulation technique described in [1]. Given the specification of a Boolean function, the package will detect all partial and total symmetries and return the appropriate multi-dimensional hyper-linear structure. This structure can then be probed for the purpose of generating simulation code. This version of the Hyper-Linear package has the tweaks necessary for detecting conjugate symmetry. Use of conjugate symmetry is the default, but this can be suppressed by using lower-level function calls.

2 Introduction
Everyone is familiar with the concept of a hyper-cube. A hyper-linear structure is the analog of a hyper-cube. The main difference is that there can be more than two vertices along each dimension. When there are three or more vertices along a particular dimension, this dimension represents a combined variable. A combined variable is simply one or more variables whose values are taken together. It is not possible to combine variables unless they are symmetric.

As in a hypercube, each dimension represents a variable. For a simple variable, traveling along the dimension takes me from a vertex where the variable has value zero to a vertex where the variable has value one, and vice-versa. For a combined variable, each vertex represents all input combinations of a certain weight. Moving along a dimension moves us from a vertex of lower weight to a vertex of higher weight, and vice-versa.

For example consider the following hyperlinear structure which represents a totally symmetric function in four variables. Since all inputs are symmetric, the hypercube representing the function can be collapsed into a one-dimensional hyperlinear structure with the single dimension representing a combination of the four input variables.

```
0 1 2 3 4
A B C D E
0000 0001 0011 0111 1111
0010 0110 1110
0100 1010
1000 1100
```

Each state in the one-dimensional structure represents all vectors of a certain weight. The values A-B are the values of the function for inputs of the corresponding weight.
When a function is partially symmetric, a multi-dimensional structure is required, with one dimension representing each partial symmetry. The construction of these structures is identical to the illustration, with each state representing the weight of the values that appear on the combined inputs.

To construct a hyperlinear structure it is necessary to detect partial and total symmetries. This is done by examining the appropriate diagonals of the hyper-cube or hyperlinear structure and collapsing variables together two at a time into linear structures with more than two states. The process starts with a zero-dimensional one-state machine. This single state contains the representation of the function. Several different forms are allowed, Truth-Tables, Logic-Diagrams, List of Code points, and Sum-of-Products expressions. The list of forms can easily be expanded.

The next step is to specialize the function by setting input 1 to zero. Then the function is specialized again by setting input 1 to one. These two functions are placed in a one-dimensional two-state state machine. The two functions are then specialized by setting input 2 to 0 and 1 in both of the new function representations. This gives four function representations that are placed in a two-dimensional four-state machine. Once this has been done, the symmetry detection algorithms are invoked.

A single symmetry detection algorithm is used, but several different types of symmetry can be detected by altering the indexing function of the state machine. To detect skew symmetries where one input is inverted with respect to the other, the indexing function is modified to access one dimension in reverse order. For conjugate symmetries where one variable is conditionally inverted by another, the indexing function is modified to access odd-numbered planes in reverse order. When a skew or conjugate symmetry is detected, the modification of the hyperlinear structure caused by the unconditional or conditional inversion is undone during the collapse.

Once all symmetries have been detected, the next variable is specialized out of the remaining functions, the existing structure is duplicated, and the specialized functions are inserted into the two copies of the original machine. These two copies are combined into a machine one dimension larger than the original. Then the detection process begins anew. This expand/collapse cycle repeats until no unspecialized variables remain, and all symmetries have been detected. The result is a multi-dimensional hyperlinear structure with function values in each node.

If skew symmetries have been detected, a list of inverted variables is maintained to document the corrective actions that have been applied to various dimensions. When conjugate symmetries are detected, an nxn matrix is saved documenting the conditional corrections that have been applied to the hyperlinear structure. The matrix and the list of inverted variables can be used by a simulator to determine how to correctly handle the hyperlinear structure.

Of course, for details, you've got to look at the references.

3 The classes

This report is not particularly concerned with the theory of how the whole thing works. We are primarily concerned with the interface. The basic hyperlinear structure is defined by the class Hyper, defined in the file “hyper2.h”. The state machine is stored as a one-dimensional array and is accessed using a protected embedded indexer class. The class provides no public variables. The following public functions are available.
3.1 Hyper

Hyper() Default constructor creates a zero-dimensional one-state machine with no function representation.

Hyper(const Hyper &X) Standard copy constructor.

Hyper(BoolRep *X) BoolRep is an abstract class describing a general function representation. This constructor creates a zero-dimensional one-state machine with X as the value of the state.

~Hyper() Standard destructor.

Hyper &operator=(const Hyper &X) Standard assignment overload.

int GetDimension() Return the number of dimensions.

int GetLength(int Dim) Return the number of states along a specified dimension.

int GetInputCount() Get the number of inputs. (This value is embedded in the BoolRep object, if available.)

int Collapsible(int x, int y) Returns 1 if dimensions x and y represent variables symmetric with one another. The test uses the current indexing mode.

void Expand(int Pos) Specialize the input variable in position Pos for both 0 and 1, double the size of the structure, and increase the dimension by 1.

void Collapse(int x, int y) Collapse dimensions x and y together into a single dimension representing a combined variable. Doing this when the dimensions are not actually collapsible will produce random garbage.

void CollapseAny(void) Test all dimensions in all indexing modes and collapse the first thing found.

int CollapseAny(int x) Test all dimensions against the specified dimension in all indexing modes and collapse the first thing found, if any.

int CollapseAny(int x, int y) Test dimensions x and y in all indexing modes and collapse the first thing found, if any.
void FullCollapse(void) Specialize the variables one at a time and keep collapsing until all variables have been eliminated and all symmetries are found. This function doesn’t necessarily do this in the most intelligent way.

int GetCubeSize() Return the number of states in the cube.

int GetCellValue(int *Dim) Usable only after full collapse when the states contain Boolean values rather than function descriptions. The argument is an indexing array that must be the same size as the number of dimensions. A value must be specified for each dimension to select a particular cell.

void SetSkew(int x) Set the indexing mode to treat dimension x as a variable that is inverted with respect to whatever variable it is compared against.

void ClearSkew() Remove the effects of SetSkew.

int GetSkew() Returns the value of the skew variable, or -1 if there isn’t one.

void SetCI(int Inverter, int Inverted); Set the indexing mode to detect a conditional inversion of one dimension by another. SetCI(1,2) is decidedly NOT the same as SetCI(2,1).

void ClearCI() Clear the effects of SetCI.

int GetCIInverter() If a conditional inversion is not set, this returns -1. Otherwise it returns the number of the dimension that DOES the conditional inverting.

int GetCIInverted() If a conditional inversion is not set, this returns -1. Otherwise it returns the number of the dimension that GETS the conditionally inverted.

BoolRep * GetBoolRep(int *Dim) Return the current Boolean Function representation contained in the state indexed by Dim. Dim must be an array that is the same size as the dimension, and must contain one integer value for each dimension.

int isInverted(int Dim) Used after a full collapse to determine whether there is a skew symmetry with respect to a particular input. This state can toggle back and forth during a full collapse.

int isConjugate(void) Returns 1 if conjugate symmetry is detected.
int GetInputDimension(int Dim)  Returns the dimension of the hypercube that corresponds to a particular input variable.

int GetMatrixValue(int x, int y)  If conjugate symmetry has been detected, this function can be used to read out the values of the matrix. This is usually done column by column.

void DumpCube()  A debugging function that is used to print the values of a fully collapsed hyperlinear structure.

3.2 \textbf{BoolRep}

The Hyper class uses the BoolRep class to perform most of its functions. It relies on a set of abstract functions to perform its tasks. BoolRep has no public variables. The public functions are as follows.

\textbf{BoolRep() } Default constructor, does nothing.


\textbf{~BoolRep() } Virtual destructor, does nothing.

\textbf{BoolRep &operator=(const BoolRep &X) } Standard assignment overload. Will work in the following way: \(*\mathit{X} = \mathit{Y};\) but only if \(\mathit{X}\) and \(\mathit{Y}\) point to elements of the same derived type.

\textbf{BoolRep * Specialize(int Pos, int Value) } Abstract. Returns a new BoolRep of the derived type, with the variable in position “Pos” set to the constant “Value”.

\textbf{int operator==(BoolRep *X) } Abstract. Equal test overload.

\textbf{int operator!=(BoolRep *X) } Abstract. Not-equal test overload.

\textbf{int GetInputCount() } Returns the input count. Must be set through derived class functions.

\textbf{BoolRep * Duplicate() } Abstract. Duplicates current object and returns a pointer to new object.

\textbf{int GetValue() } Abstract. If the BoolRep has been specialized down to a constant, or was created that way, this function can be used to return the constant.
3.3 BoolRep Derivatives

Although the BoolRep interface is all that is used by the Hyper class, most of the work of creating and maintaining the BoolRep objects must be done through the functions of the derived classes. Obviously there is no limit to the number of derived classes. Currently we provide the following.

TruthTable Implements truth tables for functions with no more than 6 inputs.

BigTruthTable Implements truth tables for functions with any number of inputs.

CodePoints Allows functions to be specified as a set of Boolean Vectors. These vectors represent the one’s of the function.

LogicDiag Allows functions to be specified as a network of gates. (Beta version)

SOP Allows functions to be specified as a sum of products Boolean Expression

We will go through the interfaces of these classes one by one.

3.4 TruthTable

The class TruthTable has no public variables. The public functions are as follows.

TruthTable() Default constructor. Creates a zero-input zero-valued function.

TruthTable(const TruthTable &X) Standard copy constructor.

TruthTable(int NewInputCount,unsigned long long NewTable) Internally, a truth table is represented as a 64-bit integer. If there are fewer than 6 inputs, the high-order bits of the integer are unused and must be set to zero. The high order bit represents the value for input 000000. This function supplies a new input count and a new truth table. The variable NewTable is not altered in any way before internal storage.

TruthTable(NETWORK *Inw) This function creates a truth table out of a network of gates. The network of gates must be obtained from the conllib package, which is available from the Baylor University Dept of Computer Science tech report archive.

~TruthTable() Standard destructor (virtual).
TruthTable &operator=(const TruthTable &X)     Standard assignment overload.

BoolRep * Specialize(int Pos,int Value) Will set the specified input to one or zero and return the truth table for the new function.

int GetValue() Returns the value for zero-input tables. This value can be zero or one.

int operator==(BoolRep *X) Equals test overload.

int operator!=(BoolRep *X) Not-equals test overload.

unsigned long long GetTable() Retrieve the truth table as a 64-bit integer.

BoolRep * Duplicate() Create a copy of the current object and return a pointer to it.

3.5 BigTruthTable

The class BigTruthTable has no public variables. The public functions are as follows.

BigTruthTable() Default constructor. Creates a zero-input, zero-valued function.

BigTruthTable(const BigTruthTable &X) Standard copy constructor.

BigTruthTable(int NewInputCount,unsigned long long * NewTable) Internally a big truth table is a collection of 64-bit integers. This function creates a table out of an array of such integers. If the number of inputs is less than 6, a single 64-bit integer must be supplied, with the actual table values in the low-order bits. The unused high-order bits must be set to zero. For 6 or more inputs, the table must contain \(2^{n-6}\) 64-bit integers. The high order bit of integer [0] corresponds to an input of all zeros.

BigTruthTable(NETWORK * Inw) This function creates a big truth table out of a network of gates. The network must be supplied by the connlib package.

~BigTruthTable() Standard destructor.

BigTruthTable &operator=(const BigTruthTable &X) Standard assignment overload.

BoolRep * Specialize(int Pos,int Value) Sets the selected input to one or zero, and returns the truth-table for the new function.
int operator==(BoolRep *X) Standard equality-test overload.
int operator!=(BoolRep *X) Standard inequality test overload.
unsigned long long * GetTable() Returns a pointer to the internal array of 64-bit integers.
BoolRep * Duplicate() Creates a duplicate of the current object and returns a pointer to it.
int GetValue() For zero-input functions, this function returns the value, which can be one or zero.
void Print() Debugging function to print the internal table.

3.6 CodePoints
The file CodePoints.h contains two classes, CodePoint and CodePoints. The CodePoints class is the primary class that is derived from BoolRep. The CodePoints class is an auxiliary class that is used to create a CodePoints object. The class CodePoint can be used independently of CodePoints, although we do not encourage this usage. We document the interface to both classes here. The classes CodePoints and CodePoint are currently being used by a number of under-development projects and are subject to change without notice.

CodePoints(int X) Creates a zero-input function with the specified value. Only the low-order bit of X will be used.
virtual ~CodePoints() Standard destructor.
virtual BoolRep * Specialize(int Pos,int NewValue) Sets the specified input to zero or one and returns the representation of the new function.
virtual int operator==(BoolRep *X) Standard equality-test overload.
virtual int operator!=(BoolRep *X) Standard inequality test overload.
CodePoints operator+(const CodePoint &X) Adds the specified code-point to all code points of the representation. Each code point is treated as a
GF(2) vector. The vectors are added (XOR) element-wise.

CodePoints &operator+=(const CodePoint &X)  
Same as +, but modifies the current object instead of returning a new object.

CodePoints operator+(CodePoints &X) 
Computes the union of two sets of code points.

CodePoints &operator+=(CodePoints &X)  
Same as +, but modifies the current object instead of returning a new object.

CodePoints operator-(CodePoints &X) 
Computes the set difference of two sets of code points.

CodePoints &operator-=(CodePoints &X)  
Same as -, but modifies the current object instead of returning a new object.

CodePoints operator*(CodePoints &X) 
Computes the intersection of two sets of codepoints.

CodePoints &operator*=(CodePoints &X)  
Same as *, but modifies the current object instead of returning a new object.

CodePoint *GetItem(int i) 
Returns a pointer to code point number i, if any. Otherwise NULL is returned. The first code point is number zero.

CodePoint *GetFirst(void) 
Gets the first code point in the list. Returns NULL if there are no code points.

CodePoint *GetNext(void) 
Gets the next code point. Used with GetFirst to enumerate all code points in the set. Returns NULL when the list is exhausted, or if GetFirst has never been called.

int GetPointCount(void) 
Return the number of points in the set.

virtual BoolRep * Duplicate() 
Duplicate the current object and return a pointer to it.

void Sort(void) 
Sort the set of points into ascending order. Each point is treated as a binary number for the purpose of sorting. Item zero of the code point corresponds to the high-order bit.

void SetInputCount(int NewInputCount) 
Sets the input count to a new value. Use with care.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Add(CodePoint X)</td>
<td>Add a new code point to the collection. The code point is copied, rather than moved to the collection. If the collection is empty, the input count of the point is copied to the input count of the collection.</td>
</tr>
<tr>
<td>void Add(unsigned char *X, int NewSize)</td>
<td>Creates a new code point with NewSize elements and adds it to the collection. If the collection is empty, NewSize also becomes the input count of the collection.</td>
</tr>
<tr>
<td>void Print(void)</td>
<td>Debugging function to print the contents of the set.</td>
</tr>
<tr>
<td>virtual int GetValue(void)</td>
<td>If the collection is empty, (thus representing a constant function), A value of one or zero is returned.</td>
</tr>
<tr>
<td>void RmDups(void)</td>
<td>Remove all duplicate code points from the collection.</td>
</tr>
<tr>
<td>int HasDups(void)</td>
<td>Check to see if the collection has duplicates.</td>
</tr>
<tr>
<td>int HasZero(void)</td>
<td>Check to see if the collection contains the zero point.</td>
</tr>
<tr>
<td>int Contains(CodePoint &amp;X)</td>
<td>Check to see if the collection contains a certain point.</td>
</tr>
<tr>
<td>int Contains(CodePoint *X)</td>
<td>Same, but allows direct use of pointers.</td>
</tr>
<tr>
<td>CodePoints GetBasis(void)</td>
<td>Obsolete. If the collection constitutes a linear vector space closed under vector addition, then this function will return a sub-collection consisting of a basis for the linear vector space.</td>
</tr>
</tbody>
</table>

**The CodePoint interface.**

The CodePoint class has a single public variable, “Next”. This variable is a pointer to a CodePoint and is used to create singly-linked lists of CodePoints.

CodePoint has the following public functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodePoint()</td>
<td>Default constructor. Creates a zero-sized element with no data.</td>
</tr>
<tr>
<td>CodePoint(const CodePoint &amp;X)</td>
<td>Standard copy constructor.</td>
</tr>
</tbody>
</table>
CodePoint(unsigned char *X, int NewSize) Creates a code point of size NewSize from the array X.

CodePoint(int *X, int NewSize) Creates a code point of NewSize from the array X.

virtual ~CodePoint() Standard destructor.

CodePoint &operator=(const CodePoint &X) Standard assignment operator overload.

int operator==(const CodePoint &X) Comparison operator overloads. Code points are treated as binary integers, with item 0 being the high-order bit. The first differing bit in ascending-index order determines the result of the < type comparisons.

int operator!=(const CodePoint &X)
int operator<(const CodePoint &X)
int operator<=(const CodePoint &X)
int operator>(const CodePoint &X)
int operator>=(const CodePoint &X)

CodePoint operator+(const CodePoint &X) Add two points. Each point is treated as a GF(2) vector and the vectors are added element-wise, using the XOR operator.

CodePoint &operator+=(const CodePoint &X) Same as +, but the current object is modified instead of producing a new object.

int GetSize(void) Returns the number of elements in the code point.

unsigned char GetItem(int Pos) Returns the item in the specified position, if any. The first element is item zero. If the index is bad, zero is returned.

void SetSize(int NewSize) Changes the size of the code point, and sets all elements to zero. This function call is ignored unless the new size is actually different from the old size. Setting a code point to size zero is not permitted.

void SetItem(unsigned char X, int Pos) Set the value of item Pos to X. The first item is item zero. If the index is bad, the function call is ignored.

void SetAll(unsigned char *X, int NewSize) Change the size of the point to NewSize, and use the array X to assign a value to all elements of the code point.

void SetAll(int *X, int NewSize) Same as above, except with integer inputs.

void Print(void) Debugging function to print the code point on standard out.
int isZero(void)  Returns 1 if all elements of the code point are zero, 0 otherwise.

long long GetInt( )  Converts the code point into a 64 integer and returns the value. The low-order bit is the element with the highest index. If there are more than 64 elements, those with the lowest indices will be thrown away.

3.7 SOP

The file SOP.h contains two classes, SOP which is derived from BoolRep and SOPItem which is used to construct SOP objects.

The SOP class has no public variables. It has the following public functions.

SOP( )  Default constructor. Creates an empty expression. This

SOP(const SOP &X)  Standard copy constructor.

SOP(int NewInputCount,SOPItem *NewSOPList)  Create an expression from an input count and a linked list of terms.

~SOP( )  Standard destructor.

SOP &operator=(const SOP &X)  Standard assignment overload.

BoolRep * Specialize(int Pos,int Value)  Sets the specified input variable to either 1 or 0, and returns the resultant expression.

int operator==(BoolRep *X)  Standard equality-test overload.

int operator!=(BoolRep *X)  Standard inequality test overload.

BoolRep * Duplicate( )  Duplicate the current object and return a pointer to the new object.

int GetValue( )  If the expression represents a constant function, return the value of the function.

void Scrub( )  First, each term is scrubbed. This process eliminates duplicate literals, and sets all terms containing logical contradictions (the same input in both primed and unprimed form) to constant zero. Once all terms are scrubbed, the collection is scrubbed. Insane terms (those referencing non-existent variables) are deleted. Constant terms are eliminated, unless there is only one. A constant zero is deleted, a constant 1 is
retained and everything else is deleted. Duplicate terms are eliminated. Useless terms (those that completely contain another term) are deleted. Collapsible terms are combined until no more remain. An example of two collapsible terms is \(abc\) and \(a'bc\) which collapse to \(bc\). This will not produce a minimal expression, because once two terms are collapsed, they are eliminated and not compared to other terms. Thus the collection \(abc, a'bc, abc'\) will be reduced to \(bc, abc',\) not \(bc, ab\).

void Print( ) Debugging function to print the collection of terms.

void Add(int *NewVList, int NewSize) Add a new term. A term consists of a collection of integers indexing the inputs of the function. The first input is input zero.

void Add(char *CSVList, int NewSize) Add a new term in the form 1,2,3, etc. The string must be a string of integers separated by zero.

void SetInputCount(int NewInputCount) Set the number of inputs to the function.

**The SOPItem Interface**
The SOPItem class has a single public variable, "Next". This variable is of type pointer to SOPItem, and is used to create linked lists of SOPItems. Each SOPItem represents one term of an SOP expression. The term is a collection of literals (primed and unprimed variables), which are assumed to be ANDed together to compute the value of the term. A collection of SOPItems is assumed to be a collection of terms that are ORed together to compute the value of the function. The AND and OR operators are not actually present.

SOPItem( ) Default constructor. Creates a constant zero term.

SOPItem(const SOPItem &X) Standard copy constructor.

SOPItem(int *NewVList, int NewSize) Creates a term out of a list of literals. Positive values are assumed to be unprimed variables, negative values are assumed to be primed variables. The one’s complement is used to create primed variables rather than ordinary negation. This permits a representation of negative zero. The magnitude indicates the position of the input variable in the input list, with the first variable having position zero. If the input variables are \(abc\), then 0x00000000 represents \(a\), 0xffffffff represents \(a'\), 0x00000001 represents \(b\), 0xffffffff represents \(b'\), and so forth.
SOPItem(int ConstVal) Creates a constant term of either 1 or 0. Only the low-order bit of Val is used.

SOPItem(char *CSVList,int NewSize) Creates a term out of a comma-separated list. Examples of input strings are $ab,c$ and $a,b',c$.

~SOPItem( ) Standard destructor.

SOPItem &operator=(const SOPItem &X) Standard assignment operator.

int operator==(const SOPItem &X) Comparison operators. First the lengths of the terms are compared. Terms with fewer literals are unconditionally smaller than terms with more literals.
int operator!=(const SOPItem &X) For terms with an equal number of literals, the first differing literal is compared. The terms are sorted into ascending order as unsigned numbers, so the primed terms follow all unprimed terms.
int operator<(const SOPItem &X) int operator<=(const SOPItem &X) int operator>(const SOPItem &X) int operator>=(const SOPItem &X) If a term is completely contained in another, it implies that term. Thus $ab$ implies $abc$, but not $ab',c$.

SOPItem * Specialize(unsigned int VarId,int Value) Set the specified input to either zero or 1 and return the new term. Setting a literal (as opposed to an input) to 0 returns a constant 0. Setting a literal to 1 returns a term with that literal missing.

SOPItem * CollapseWith(SOPItem * Candidate) If the candidate is collapsible with the current object, then a new term is created with the collapsible literal eliminated. For a candidate to be collapsible, it must be identical with the current object, except for one literal. The differing literal must be primed in one term and unprimed in the other. The result is a copy of the current object with the differing literal erased. Non-collapsible terms cause this function to return NULL.

int GetValue( ) If the term is constant, this function returns the value of the constant. Non-constants return 0.

void Print( ) Debugging function to print the term.

int GetSize( ) Return the number of literals in the term.

int GetItem(int x) Return the specified literal. If x is invalid, zero is returned.
int isConstant() Returns 1 if the term is constant, 0 otherwise.

int isSane(int InputCount) Returns 1 if the term contains any literal referencing a
variable numbered greater than or equal to InputCount.

3.8 LogicDiag (Beta)
This collection of objects is still being developed. Some things work, some things don’t. If
you use it, please report any bugs. If you fix any bugs, send me the fix.

The LogicDiag class has no public variables. It has the following public functions.

LogicDiag( ) Default constructor. Creates a null logic network. This will
generate errors with some of the other functions.

LogicDiag(const LogicDiag &X) Standard copy constructor.

LogicDiag(BETTERNETWORK *NewNw) This is the constructor you should use. Suppose you
have the output from the connlib package, and that the
variable nwhead points to this output. Use the following
declaration:
LogicDiag X(MakeBetter(nwhead));

~LogicDiag( ) Standard destructor.

LogicDiag &operator=(const LogicDiag &X) Standard assignment overload.

BoolRep * Specialize(int Pos,int Value) Set an input variable to zero or one and compute and
return the resultant new network. This function has known
bugs.

int operator==(BoolRep *X) Standard equality overload.

int operator!=(BoolRep *X) Standard non-equals overload.

BoolRep * Duplicate() Duplicates the current object and returns a pointer to the
duplicate.

BETTERNETWORK *GetNetwork(void) Primarily a debugging function. Returns a pointer to
the network being held by the object. The next step would
be to print this network using the following function.

void PrintBetterNetwork(BETTERNETWORK *Inw,FILE *ifile);

int GetValue( ) Returns the value of a constant network.
The two global functions MakeBetter and PrintBetterNetwork are available in the Hyper-Linear library.

4 Installing the library

All of this stuff is available in the hyper-linear library. To install this package, download the hyper2.zip file from the Baylor Technical Report Archive:

https://beardocs.baylor.edu/handle/2104/4824//browse-title

Alternatively, you could start here:

http://beardocs.baylor.edu

and navigate down through the school of engineering and computer science, to the department of computer science, to the department of computer science technical report collection and then browse by title or whatever.

This is part of the FHDL software collection, as is every piece of software with author Peter M. Maurer.

Create a separate directory for the collection, and create three sub-directories: bin, include, and lib. These packages will compile under Linux/gcc and under Visual C++. They should also compile under Dev-C++ or any other windows version of gcc, but this has not been tested.

Unzip the package into your FHDL collection directory. This will give you a directory named “hyper2”. Don’t create this directory yourself, just let the unzip process create it for you. Included with the software are a Makefile and a collection of Visual C++ project files. You can use these files to compile the library. The make process will also create an executable file, but this is useful only for debugging purposes, and you can ignore it. The .h files in your hyper2 directory should be copied into your include directory, and the result of the compilation should be copied into your lib directory. The copying will be done automatically under Linux. The Linux file name is “libhyper2.a”. Under Visual C++, the name of the library is hyper2lib.lib. This file will end up in a directory under your hyper2 directory. This directory will be named “Debug” or “Release” depending on which mode you use for compilation. (I recommend “Debug” mode.) This file should be copied to your lib directory.

If you use Visual C++, your life will be considerably easier if you make Visual C++ aware of your lib, include and bin directories. This is done by starting with the “Tools” menu and selecting “Options …”. Then go to “Projects and Solutions” sub-menu item VC++ Directories. Add “bin” to the executable files list, “include” to the include files list, and “lib” to the library files list. To use the package, include the file “hyper2all.h”, and add “hyper2lib.lib” as an additional linker input file in your project options. In most cases you will also need to add “connlib.lib” as a linker input file.

This package requires prior installation of the connlib package for proper operation.
5 References.