Appendix I – NSLM Methods

We describe in this appendix a number of library methods in addition to those already introduced in chapter 6.

A.I.1 System Methods

NSLM provides a number of system methods for getting and setting the NSL system variables and for manipulating the simulation. These methods are accessed using the “system” prefix as follows:

```
   system.methodCall();
```

We mention the most important system methods in the following sections. Note that chapter 7 commands that begin with the “nsl” prefix all have an equivalent method call in NSLM.

In general, setting parameter values at the system level may be overridden by each module, i.e. system method calls to set parameter value are used as a default, such as setting the value for a “delta” for the complete system while each module may assign its own particular value.

Data Access

The **nslSetAccess** method sets the default NSL access of the entire system:

```
   system.nslSetAccess('W');
```

This is an important statement since all model variables get their default access from the system, similarly all module variables get their default access from either the model or their corresponding parent modules and so on. The **nslSetAccess** method takes a single character as argument, either ‘W’, ‘R’ or ‘N’ for write/read, read, and no-access, respectively. The system default setting is ‘W’. (We hope to change the default access to ‘R’ in a future version.) The **nslGetAccess** method will retrieve the default access value for the system,

```
   char cur_access = system.nslGetAccess();
```

Simulation Parameters

Simulation parameters are usually set from a model’s **initSys** method. It is important to remember that simulation time starts at 0, cycles start at 1, and epochs start at 1. Code segment A.I.1. shows the most important simulation parameters that may be set at the system level,

```
   system.setTrainEndTime(dval);
   system.setRunEndTime(dval);
   system.setTrainDelta(dval);
   system.setRunDelta(dval);
   system.setNumTrainEpochs(ival);
   system.setNumRunEpochs(ival);
```

Code Segment A.I.1: Methods for setting simulation parameter values for the entire system where dval corresponds to a double value and ival corresponds to an int (integer) value.

Getting the simulation parameters with a **double** or **int** type *var* as shown in the following methods in code segment A.I.2.

```
dval = system.getTrainEndTime();
dval = system.getRunEndTime();
dval = system.getTrainDelta();
dval = system.getRunDelta();
ival = system.getNumRunEpochs();
ival = system.getNumTrainEpochs();
ival = system.getCurrentTime();
ival = system.getCurrentCycle();
ival = system.getCurrentEpoch();
ival = system.getTrainEpoch();
ival = system.getRunEpoch();
```

Code Segment A.I.2: Methods for getting simulation parameter values from the entire system where *dval* corresponds to a **double** value and *ival* corresponds to an **int** (integer) value.

Incrementing, Breaking and Continuing

This section describes methods for incrementing, breaking and continuing with system defined loops. Note that unless otherwise specified, the method can be applied in either the training phase or the run phase. Methods that increment counters are:

```
system.incCycle();
system.incRunEpoch();
system.incTrainEpoch();
system.incTime();
```

Methods that break the simulation between modules, cycles, or epochs are:

```
system.breakModules();
system.breakCycles();
system.breakEpochs();
```

Methods continuing with the next module, cycle, or epoch after a break are:

```
system.continueModule();
system.continueCycle();
system.continueEpoch();
```

Model Variables

NSL lets the user set and get a number of parameters from existing model variables.

To set the name of an object instance we use:

```
obj.nslSetName(charString);
```

To get the NSL instance name of an module or class object we use:

```
charString name=obj.nslGetName();
```

The user may obtain or assign values to and from arbitrary variables in a model using the **nslSetValue** and **nslGetValue** methods, respectively. Note that all value setting and getting using these functions requires a corresponding data access similar to NSLS script data accessing. To set the value of a variable *var1* to a variable *var2* the user may use the following functions (*cast* represents a NSLM object type where variables, *var1* and *var2*, are object types as well),

```
var1=(cast)system.nslGetValue(“var1”);
system.nslSetValue(“var2”, var1);
```

Note that in the above methods the user must know each variable’s absolute name starting at the tree hierarchy root, i.e. the model name. For example, if we wanted to get the value of a variable named *w1* located in module *m1* of model *modelA* and assign it to a variable *foo*, we would type,

```
NslFloat1 foo(10);
foo=(NslFloat1)system.nslGetValue(“modelA.m1.w1”);
```

or

```
NslFloat1 foo(10);
system.nslSetValue(“modelA.m1.w1”, foo);
```

In some cases the user may want to convert some of the above object type values into primitive types. This is accomplished using the **nslGetValue** method applied directly to an object type. This applies only to scalar object types using an appropriate cast. For example for a scalar **NslFloat0** *w0* object we could do the following,

```
NslFloat0 doo;
doo=(NslFloat0)system.nslGetValue(“modelA.m1.w0”);
double d;
d=(double)doo.nslGetValue();
```

Note that simple assignment wouldn’t work since a primitive type cannot be directly assigned from an object type unless such method is present.

There are a number of methods that obtain values from objects of higher dimensions. The **getDimensions** method returns an integer specifying whether the object has 0, 1, 2, 3, or 4 dimensions.

```
int dim=obj.getDimensions();
```

The **getSizes** method obtains the different dimensions of an object where *obj1* is of dimension 1, *obj2* is of dimension 2 and so on:

```
obj1.getSizes(int);
obj2.getSizes(int,int);
obj3.getSizes(int,int,int);
obj4.getSizes(int,int,int,int);
```

We can also get the size of each dimension individually.

```
int size1=obj1.getSize1();
int size2=obj2.getSize2();
int size3=obj3.getSize3();
int size4=obj4.getSize4();
```

One more note about the different *getSizes* methods is that they can be used to control the looping for say initialization of a variable. For example, to assign a value to each element in the two-dimensional NSL object we could use the **for** control statement as follows,

```
NslFloat2 y(2,3);
int i,j;
for (i=0; i<y.getSize1(); i++)
   for (j=0; j<y.getSize2(); j++)
      y[i][j] = i+j;
```

where the functions *y*.**getSize1**() and *y*.**getSize2**() get the *rows* and *columns* sizes, respectively.

We include as well the following methods returning different sectors of multidimensional objects. For example, to get the jth column (1 dimensional object) of a 2 dimensional object we would do:

```
obj2.nslGetColumn(j);
```

(Recall that a row is simply obtained using squared brackets, e.g. obj[*i*].) To obtain a 2 dimensional sector from a 2 dimensional object we would do

```
obj2.nslGetSector(start1, start2, end1, end2);
```

Similarly, for 3 dimensional objects we would do:

```
obj3.nslGetSector(start1, start2, start3, end1, end2, end3);
```

For a 4 dimensional object we would do:

```
Obj4.nslGetSector(start1, start2, start3, start4,
   end1, end2, end3, end4);
```

Dynamic Memory Allocation

As introduced in chapter 6 NSL lets the user set the size of an object in a dynamic fashion. This applies to other than scalar types having dimensions higher than 1. The user first instantiates the variable as follows without specifying its actual size:

```
VisibilitySpec ObjectType obj();
```

Then a call to the dynamic memory allocation routine is done where *sizeList* depends on the particular *ObjectType* chosen:

```
obj.nslMemAlloc(sizeList);
```

For example, in chapter 18, the “Face Recognition by Dynamic Link Architecture” model defines almost all the variables in such a way as *NormFactor*:

```
private NslFloat2 normFactor();
```

Since the dimensions of the variable type is 2 then two arguments are passed to the memory allocation routine:

```
normFactor.nslMemAlloc(i1max,j1max);
```

Printing

Printing data in NSL takes the form of **nslPrint** and **nslPrintln** (print on a new line) for output of any string or variable. Note that we do not preface them with “system”. For example, to print the value of a variable we would do (note the use of the “+” string concatenation operator),

```
nslPrint("x="+ x);
```

The above represents an implicit conversion from any variable type into a string equivalent to the explicit form:

```
nslPrint("x="+ x.toString());
```

Additionally, to print the name of an object we type:

```
nslPrint("x="+ x.getName());
```

Since every class should have a "**toString**" method, we provide one for the "system". The system **toString** method returns the current model name if it is set and an error string if it is not set:

```
nslPrintln(system.toString());
```

Another useful method is **nslPrintAllVariables**. This method prints out the name and value of all variables in the system; however, this method is very time consuming and we recommend using it sparingly.

```
system.nslPrintAllVariables();
```

Also the **nslPrintStatistics** is very useful, printing the current model name, the current phase (initialization, train, run, or end), the current epoch, the current time and the current cycle.

```
system.nslPrintStatistics();
```

File Manipulation

NSL supports reading and writing into external files.[1] NSLM defines a **NslFile** object class for doing the corresponding input and output manipulations. For example, to access a file named "file.dat" (suffix is not relevant), the user must first define an object holding the reference to the file as follows,

```
NslFile file("file.dat");
```

To open a file we use the following function specifying the type of interaction we want to use: '**R**' for read only, '**A**' (all) for both read and write or '**W**' for write only. For example, to opening "file.dat" for both read and write,

```
file.open('A');
```

To close the file we simply do,

```
file.close();
```

Also since the text files we use are buffered, we provide the flush command to immediately flush the buffer:

```
file.flush();
```

To write string values into the file one line at a time, we use the method **puts** just as in the NSL script language that is based on TCL. Note that the **puts** method write one line at a time, and will convert numerical objects to strings of characters:

```
file.puts(obj);
```

To read a value into a charString object named **obj**, we would use the method **gets** which gets one line of text and puts the whole line into **obj:**

```
file.gets(obj);
```

To write string values into the file one lexeme at a time, we use the method **write**. Note values are separated by *white-space* (space, tab, carriage return, linefeed). The method **write** will also convert numerical objects to strings of characters:

```
file.write(obj);
```

To write a value into the file with a new line at the end,

```
file.writeln(obj);
```

To read a value into a charString object named **obj**, we would use the method **read** , which gets one lexeme of text and puts the whole lexeme into **obj:**

```
file.read(obj);
```

In addition, what we have defined as *white-space*, may not be what the user desires; thus we provide two more methods that allow the user to define what *white-space* is, where *char1* specifies one character to put between lexemes and *array10* specifies a native array of 10 characters defining white-space

```
file.write(obj,char1);
file.read(obj,array10);
```

For example, the *Backpropagation* model of chapter three uses the readFile method described in the code segment A.1.3 to read training data from a file.

```
public int readFile(CharString fname,NslInt1 nPats,
NslFloat2 pInput, NslFloat2 pOutput, int iSize, int oSize)
{
   int pat=0;
   int i,j =0;
   status=-1;
   NslFile file(fname);

   if (file.open('R') <0) {
      nslPrintln("Bad File Name: "+fname);
      return(status);
   } else {
      file.gets(nPats);

      for (pat = 0; pat < nPats; pat++) {
         for (i = 0; i < iSize; i++) {
         file.read(pInput[pat][i]);
      }
      for (j = 0; j < oSize; j++) {
            file.read(pOutput[pat][j]);
      }
      file.close();
  }
}
```

Code Segment A.1.3
Example of the readFile method within the Backpropagation mode.

Display Step

The following display and protocol methods are provided for manipulation of the displays and creation and selection of the protocols. The value of a variable *t* representing the display delta or update time can be set or get as follows,

```
system.setDisplayDelta(t);
double var = system.getDisplayDelta();
```

A.I.2 Mathematical Methods

Numerical methods/functions supported by NSLM can be applied to any numerical object of any dimension (**NslInt**, **NslFloat**, **NslDouble**, dimensions 0,1,2,3,4 unless otherwise specified) or primitive type (**int**, **float**, **double**). They consist of *arithmetic*, *threshold*, *differential approximation* methods as well as some additional miscellaneous functions. Some of these functions have a corresponding operator, see section 6.2. There are two general formats for methods, the first one where the resultant is passed as return value and the second where the resultant is passed as the first parameter to the method as shown next,

```
z = method(x,y);
method(z,x,y);
```

While the first style is more elegant the second one is more efficient since a return value requires additional memory allocation, a relatively slow operation that should be avoided if possible. In particular, this becomes critical when dealing with higher level object dimensions.

Basic Arithmetic Methods and Operators

NSLM provides a number of numerical functions taking either a single or two arguments and returning a value.

- The arithmetic functions shown in table A.I.1 are defined for *x*, *y* and *z* of similar NSL type and dimensions

Operator Expression	Method Expression	Description
$z = x + y$	z = nslAdd(x,y)	element by element addition
$z = x - y$	z = nslSub(x,y)	element by element subtraction
$z = x$ ^ y	z = nslElemMult(x,y)	element by element multiplication
$z = x / y$	z = nslElemDiv(x,y)	element by element division

Table A.I.1
Basic arithmetic operators and methods.

If both operands are of the same dimension, the operation will apply to corresponding object elements. For example, in the case of 2D arrays if *x* and *y* are 4-by-4 matrices, the expression “*z*=*x*+*y*” will add elements $x[i][j]$ with $y[i][j]$ and store the resulting $x[i][j]+y[i][j]$ into $z[i][j]$, for $0 \le i,j < 4$. Additionally, if one argument of the operation is an array and the other one is a scalar, the operation will apply to every object element with the scalar number. For example, if *x* is a scalar, *y* is 4-by-4 matrix, the expression “*z*=*x*+*y*” will add element $y[i][j]$ with *x* and store the resulting $x+y[i][j]$ into $z[i][j]$, for $0 \le i,j < 4$.

- The “*” product operator works for scalars, a vector and a scalar, a scalar and a vector, a vector and a vector, a vector and a two dimensional matrix, and two, two dimensional matrices. In the case where a scalar is involved, the “*” operator will call **nslElemMult**. In the case of two vectors, then the “*” operator will call

nslElemMult as well. In the case where two matrices are involved, then the "*" operator will call **nslProduct** and return a matrix**.**

In table A.I.2 **nslProduct** assumes *x* is a matrix and *y* is matrix having the same number of rows as the number of columns in *x*. The size and dimension of resultant, *z*, is constructed from the number of rows in *x* and the number of columns in *y*.

Operator Expression	Method Expression	Description
*z = x * y*	*z* = nslProduct(*x*,*y*)	vector/matrix product

Table A.I.2
Multiplication method.

In the case where *x* is a matrix or a vector and *y* is a vector, we should use the **nslTrans** vector transpose method (see following sections) on the vector to make it a *column vector*.

The convolutions operators and methods shown in table A.I.3 are defined for both vectors and matrices of two dimensions. The type and dimension of *z* corresponds to that of *y*.

Operator Expression	Method Expression	Description
z = x @ y	*z* = nslConv(*x*,*y*)	zero-edge convolution
	z = nslConvW(*x*,*y*)	wrap-edge convolution
	z = nslConvC(*x*,*y*)	copy-edge convolution

Table A.I.3
Multiplication method.

For example, for the zero-edge effect and two matrices x, and y we have:

$$y = \begin{cases} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{cases} \qquad x = \begin{cases} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 2 \\ 1 & 2 & 4 & 4 & 4 \\ 1 & 2 & 4 & 8 & 8 \\ 1 & 2 & 4 & 8 & 0 \end{cases}$$

will result in "z = x@y"as follows:

- First a larger matrix is created with 0 values for the edges (the size of the new matrix depends on both the size of the mask and the convolved matrix; for example for a (2d+1)x(2d+1) mask, the border of zeroes must be d-deep):

$$xc = \begin{cases} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 2 & 2 & 2 & 2 & 0 \\ 0 & 1 & 2 & 4 & 4 & 4 & 0 \\ 0 & 1 & 2 & 4 & 8 & 8 & 0 \\ 0 & 1 & 2 & 4 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{cases}$$

- Second we overlap *y* on the left top corner of *xc* with *y* [0,0] on top of *xc*[0,0] so the first convolution will be given by:

```
   z[0,0] = (0*1 + 0*1 + 0*1) + (0*1 + 1*2 + 1*1) +
      ( 0*1 + 1*1 + 2*1) = 6
```

and so on for the other elements. For the wrap around edge effect and copy edge effect please see the website.

Additional Arithmetic Methods

NSL offers a number of additional arithmetic functions. We describe the most important of these in table A.I.4.

Method Expression	Description
z=nslAbs(*x*);	absolute value
z=nslDistance(*x*,*y*);	calculates the distance to a point x,y.
z=nslGaussian(*x*,*mean*,*stddev*);	*guassian* distribution for *x* with defaults of mean 0, and standard deviation 1.
z=nslRandom(*x*,*lower*, *upper*);	Calculates a random value for every element of var9 between the bounds of lower and upper. The defaults for lower and upper are 0 and 1.
z=nslRint(*x*);	rounds every element of *x* to an integer value but returns the values in the same native type of array as *x*. The variable *z* must be of the same native primitive type as the value stored by *x* (int, float, double).
z=nslExp(*x*);	calculates e to the power *x.*
z=nslLog(*x*);	calculates the *log* of *x.*
z=nslPow(*x*,*n*);	calculates *x* to the power of *n*. The variable *n* must be of the same native primitive type as the value stored by *x* (int, float, double).
z= nslSqrt(*x*);	calculates the square root of *x*.

Table A.I.4
Abs,Distance,Gaussian,Random,Rint,Exp,Log,Pow, and Sqrt methods.

For example, the **nslDistance** function calculates the distance to a point *x*,*y* using the following formula:

```
z=sqrt(pow(x,2)+pow(y,2));
```

In table A.I.5 we include different forms of the maximum and minimum methods.

Method Expression	Description
z=nslMaxValue(*x*);	finds the element with the maximum value throughout all of *x* and returns it in *z*. Variable *z* must be of the same native primitive type as the values of *x*.
z=nslMinValue(*x*);	finds the element with the minimum value throughout all of *x* and returns it in *z*. Variable *z* must be of the same native primitive type as the values of *x*.
nslMaxElem(*nj*,*x*);	finds the maximum value in a vector *x* returning the index of the element
nslMinElem(*nj*,*x*);	finds the minimum value in a vector *x* returning the index of the element
nslMaxElem(*ni*,*nj*,*x*);	finds the maximum value in a matrix *x* returning the index of the element
nslMinElem(*ni*,*nj*,*x*);	finds the minimum value in a matrix *x* returning the index of the element
nslMaxElem(,*nh*,*ni*,*nj*,*x*);	finds the maximum value in a 3d array *x* returning the index of the element
nslMinElem(*nh*,*ni*,*nj*,*x*);	finds the minimum value in a 3d array *x* returning the index of the element
nslMaxElem(*ng*,*nh*,*ni*,*nj*,*x*);	finds the maximum value in a 4d array *x* returning the index of the element
nslMinElem(*ng*,*nh*,*ni*,*nj*,*x*);	finds the minimum value in a 4d array *x* returning the index of the element
z=nslMaxMerge(*x*,*y*);	Calculates the maximum between the two elements of *x* and *y* returning it in *z*.
z=nslMinMerge(*x*,*y*);	Calculates the minimum between the two elements of *x* and *y* returning it in *z*.

Table A.I.5
Maximum and minimum methods.

Since we have up to four dimensions, the **nslMaxElem** method is overloaded and can have 2, 3, 4, or 5 parameters. For example, in the case where *x* is a four-dimensional object we use the return variables *ng*, *nh*, *ni*, and *nj* which are indexes of type **NslInt0** and are returned as well as the element they point to which is *z*. *z* must be of the same native primitive type as the values of *x*. We perform a similar task for function **nslMinElem**.

In table A.I.6 we include different forms of the sum methods.

Method Expression	Description
z= nslSum(*x*);	sums all of the values in *x* and returns it in *z*. Variable *z* must be of the same native primitive type as the values of *x*.
z= nslSumColumns(*x*);	sums the columns of matrix *x* and returns a native vector of the same type as the values of *x* and returns it in *z*. Variable *z* must be of the same native primitive type as the values of *x*.
z= nslSumRows(*x*);	sums the rows of matrix *x* and returns a native vector of the same type as the values of *x* and returns it in *z*. Variable *z* must be of the same native primitive type as the values of *x*.

Table A.I.6
Sum methods.

In table A.I.7 we include different forms of the fill method.

Method Expression	**Description**
z = nslFillColumns(*x*,*y*);	the method takes a *y* vector and fills every column of matrix *x* with it. The length of *y* and the number of rows in *x* must match. Also the values of *x* and *y* should be of similar types, and *z* should all be of the same type as *x*.
z = nslFillRows(*x*,*y*);	the method takes a *y* vector and fills every row of matrix *x* with it. The length of *y* and the number of columns in *x* must match. Also the values of *x* and *y* should be of similar types, and *z* should all be of the same type as *x*.

Table A.I.7
Fill methods.

In table A.I.8 we include different forms of the set and get method.

Method Expression	**Description**
z= nslGetColumn(*x*,*n*);	the method takes a matrix *x* and returns a vector *z* made of the *n*-th column of *x* The length of *z* and the number of rows in *x* must match. Also the values of *x* and *z* should be of similar types.
z= nslGetRow(*x*,*n*);	the method takes a matrix *x* and returns a vector *z* made of the *n*-th row of *x*. The length of *z* and the number of columns in *x* must match. Also the values of *x* and *z* should be of similar types.
z= nslGetSector(*x*,*start1*,*start2*, *end1*,*end2*);	returns the specified sector of matrix *x*. Variables *start1* through *end2* should be **int** or **NslInt0**.
z= nslGetSector(*x*,*start1*,*start2*, *start3*, *end1*,*end2*,*end3*);	returns the specified sector of 3d array *x*. Variables *start1* through *end3* should be **int** or **NslInt0**.
z= nslGetSector(*x*,*start1*,*start2*, *start3*,*start4*,*end1*,*end2*, *end3*,*end4*);	returns the specified sector of 4d array *x*. Variables *start1* through *end4* should be **int** or **NslInt0**.
nslSetColumn(*z*,*x*,*n*);	set a vector z with a vector made of the *n*-th column of *x* The length of *z* and the number of rows in *x* must match. Also the values of *x* and *z* should be of similar types.
nslSetRow(*z*,*x*,*n*);	set a vector z with a vector made of the *n*-th row of *x*. The length of *z* and the number of columns in *x* must match. Also the values of *x* and *z* should be of similar types.
nslSetSector(*z*,*x*,*start1*,*start2* ,*end1*,*end2*);	sets the specified sector of *z* with a matrix *x*. Variables *start1* through *end4* should be **int** or **NslInt0**.
nslSetSector(*z*,*x*,*start1*,*start2*, *start3*,*end1*,*end2*, *end3*);	sets the specified sector of *z* with a 3d array *x*. Variables *start1* through *end4* should be **int** or **NslInt0**.
nslSetSector(*z*,*x*,*start1*,*start2*, *start3*,*start4*,*end1*,*end2*, *end3*,*end4*);	sets the specified sector of *z* with a 4d array *x*. Variables *start1* through *end4* should be **int** or **NslInt0**.

Table A.I.8
Set and Get methods.

In table A.I.9 we include matrix transformation methods.

Method Expression	**Description**
z= nslTrans(*x*);	the method transposes a vector or matrix *x* into *z*.
z= nslInverse(*x*);	the method computes the inverse of a matrix *x* into *z*.

Table A.I.9
Fill methods.

Trigonometric Methods

In table A.I.10 we include a number of trigonometry methods.

Method Expression	Description
z= nslCos (*x*);	compute cosine of the values in *x* and returns it in *z*.
z= nslSin (*x*);	compute sine of the values in *x* and returns it in *z*.
z= nslTan (*x*);	compute tangent of the values in *x* and returns it in *z*.
z= nslArcCos (*x*);	compute arc cosine of the values in *x* and returns it in *z*.
z= nslArcSin (*x*);	compute arc sine of the values in *x* and returns it in *z*.
z= nslArcTan (*x*);	compute arc tangent of the values in *x* and returns it in *z*.

Table A.I.10
Trigonometry methods.

Threshold Methods

NSLM provides with a number of threshold functions: *ramp*, *step*, *saturation*, *bound*, and *sigmoid*, as shown in figure A.I.1. All these functions are considered *pointwise* operations similar to addition, being applied to corresponding elements in the object independent of dimension.

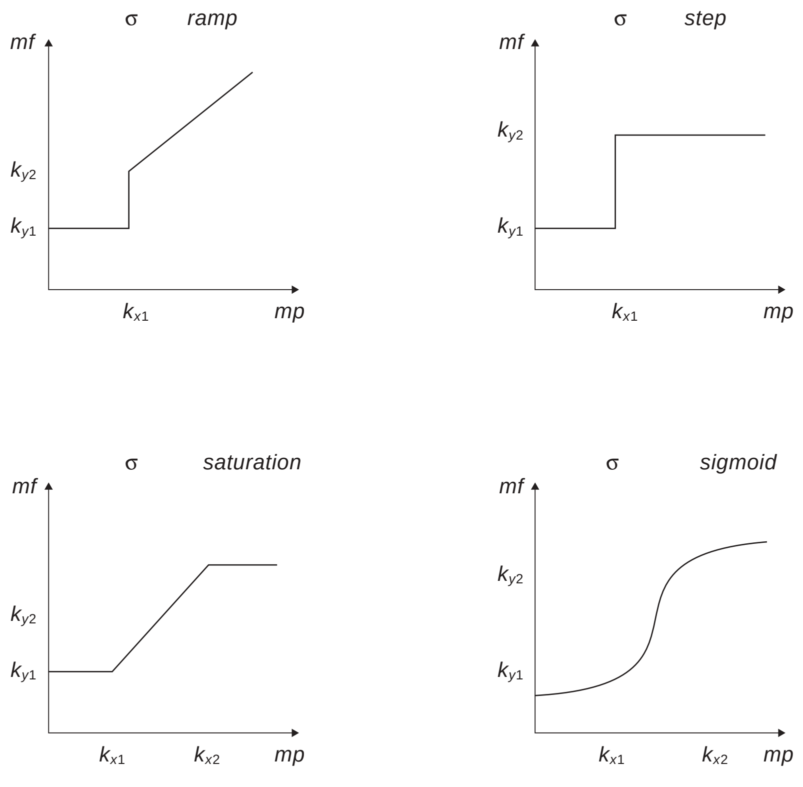

Figure A.1.2
Common Threshold Functions.

The functions are defined as follows:

- *Step* function is defined for *x* and *y* of similar NSL type and dimensions

```
y = nslStep(x,k_x1,k_y1,k_y2)
```

corresponding to the pointwise application of

$$y = \begin{cases} k_{y2} & x \geq k_{x1} \\ k_{y1} & x < k_{x1} \end{cases}$$

with defaults $k_{x1} = 0$, $k_{y1} = 0$, and $k_{y2} = 1$.

- *Ramp* function is defined for *x* and *y* of similar NSL type and dimensions

```
y = nslRamp(x,kx1,ky1,ky2)
```

corresponding to the pointwise application of

$$y = \begin{cases} x - k_{x1} + k_{y2} & x \geq k_{x1} \\ k_{y1} & x < k_{x1} \end{cases}$$

with defaults $k_{x1} = 0$, $k_{y1} = 0$, and $k_{y2} = 0$.

- *Saturation* function is defined for *x* and *y* of similar NSL type and dimensions

```
y = nslSaturation(x,kx1,kx2,ky1,ky2)
```

corresponding to the pointwise application of

$$y = \begin{cases} k_{y2} & x \geq k_{x2} \\ \frac{(k_{y2} - k_{y1})(x - k_{x1})}{(k_{x2} - k_{x1})} + k_{y1} & k_{x1} \leq x < k_{x2} \\ k_{y1} & x < k_{x1} \end{cases}$$

with defaults $k_{x1} = 0$, $k_{x2} = 1$, $k_{y1} = 0$, and $k_{y2} = 1$.

- *Bound* function is defined for *x* and *y* of similar NSL type and dimensions

```
y = nslBound(x,kx1,kx2,ky1,ky2)
```

corresponding to the pointwise application of

$$y = \begin{cases} k_{y2} & x \geq k_{x2} \\ x & k_{x1} \leq x < k_{x2} \\ k_{y1} & x < k_{x1} \end{cases}$$

with defaults $k_{x1} = 0$, $k_{x2} = 1$, $k_{y1} = 0$, and $k_{y2} = 1$.

- *Sigmoid* function is defined for *x* and *y* of similar NSL type and dimensions

```
y = nslSigmoid(x,slope,offeset)
```

corresponding to the pointwise application of

$$y = \begin{cases} \frac{1}{\left(1 + \exp^{(-slope(x - offset))}\right)} \end{cases}$$

with defaults *x*= 1, *slope* = 1, *and offset=0,* and

```
   y = nslSigmoid(x,kx1,kx2,ky1,ky2,inverseErrorConst);
```

corresponding to the pointwise application of the above function but with the following substitutions:

```
   offset=(kx1+kx2)/2;
   slope=(inverseErrorConst /(kx2-kx1));
   result=nslSigmoid(x,slope,offset) * (ky2-ky1) + ky1;
```

with defaults $k_{x1} = 0$, $k_{x2} = 1$, $k_{y1} = 0$, and $k_{y2} = 1$, error_const=10.

Notes

1. In the current version only ascii (text) files are supported. The NSLC system supports extensions to binary files as exemplified in chapter 18 with the DLM model.