Chapter 4 Notebook Calculations

Chapter Overview

The notebook window in NCSA DataScope contains a powerful data manipulation system based on two-dimensional arrays of floating-point data. This chapter describes how to use these features and describes the built-in functions that NCSA DataScope provides.

Notebook Calculations

To perform calculations using the notebook window:

1. Enter a calculation as a FORTRAN-like assignment statement, using the variable names for the currently loaded datasets that appear as the window titles (see the sample statements below).

2. Select the statement.

3. Choose Calculate From Notes from the Numbers menu or press ⌘-R. The assigned variable in the formula is created from the calculation.

Several sample assignment statements are presented in Figure 4.1:

Figure 4.1 Sample Assignment Statements

```
logden = log(density)
Ftemp = (Ctemp*5.0/9.0) + 32.0
ke = 1/2 * mass * vel * vel
magnitude = sqrt( xvel*xvel + yvel*yvel )
c = 3.1416*sin(.65)
```

Rules for Calculating

Notebook calculations may contain the operators *, / , +, –, and unary –. They may also contain parentheses and any of a selection of functions. The supported functions are described in the next section, "Built-In Functions." A *function* is a name followed by one or two parameters in parentheses, e.g., sin(x).

There are two ways to indicate the equation to be evaluated by the Calculate from Notes command:

- Select the entire equation with the mouse.

- Insert the blinking cursor on a line which contains an equation and nothing more.

If there is a selection region, NCSA DataScope uses the characters within the highlighted region. If not, it tries to use all of the characters that are on the line which contains the cursor.

All arrays used in a calculation must have the same row and column dimensions. The new variable created has the same dimensions as the arrays that were used to produce it. For example, adding the two arrays means that each element from the first array is added to its corresponding element in the second array. This produces one resulting value at each position that is placed into the array for the new variable.

When constants are combined with arrays, the constant is expanded to become an array with each element set to that constant. For example:

```
c = a + 3.2
```

is equivalent to

```
c = a + b
```

if and only if *b* is an array of the same size as *a* with each element of *b* equal to 3.2.

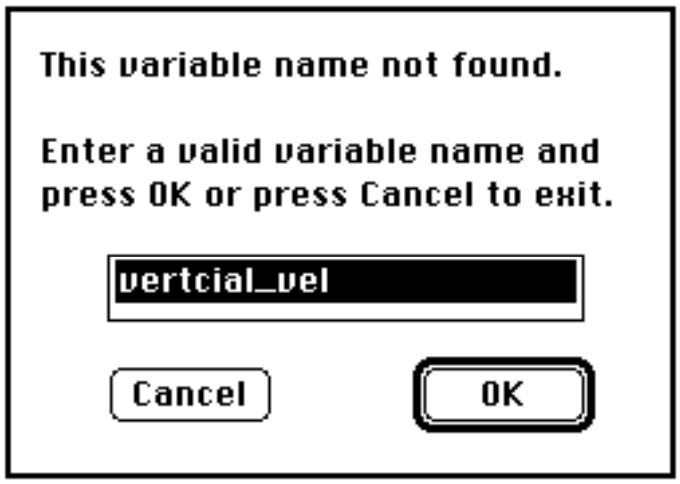
Attributes for the newly created variable are copied from the first source variable in the expression. The newly created variable has the same dimensions as the source variable. A text window is created and appears on the screen with the new variable name from the formula. The maximum and minimum values that define the region of interest are automatically calculated by finding the maximum and minimum values of the resulting array. You can change these variables in the Attributes dialog box to select a different region of interest (see "Specifying Text Window Characteristics" in Chapter 3).

Function names may be upper, lower, or mixed case; i.e., they are not case sensitive. For example, SIN, sin, and Sin may all be used to call the same sin function. On the other hand, variable names entered into equations must match the dependent variable names in the text windows exactly. For example, Density is not equivalent to DENSITY.

NOTE: The legal characters for a variable name include all of the alphanumerics and the underscore (_). Periods, backslashes, asterisks, and plus or minus signs (. / * + –) are *not* allowed in variable names. For example, pressure/dens, ke.hdf, and v*u are invalid variable names; but xv102 and K_Energy are permissible.

If you misspell a variable name or use one which is not loaded into NCSA DataScope's memory, the Variable Name dialog box (Figure 4.2) appears to question you about the spelling when you execute the equation. Enter a valid variable name and click OK or press RETURN to return to the calculation. If you click Cancel to abort the calculation and return to the notebook, an error message appears in the notebook.

Figure 4.2 **Changing the Variable Name**



Constant Result Calculations

As shown in the example presented in the preceding section, a calculation does not need to contain any variables. It may contain only constants; therefore, the formula may yield a constant value result. In this case, a text window with an array is not required. Instead, the value is returned to the notebook window. After the Calculate From Notes command is issued, the answer appears in the notebook window, as shown below.

```
x = 1.23456*1.00
***Result: 1.23456
```

The result value of the calculation is reported in the notebook directly below the selected formula. The assignment variable is ignored.

Built-In Functions

NCSA DataScope contains a wide variety of built-in functions, but it continues to develop and improve in this respect. If you need functions that are not provided, please request them so that they may be added to future releases. You may make a request in writing via U.S. mail at:

NCSA Software Development—DataScope
152 Computing Applications Bldg.
605 East Springfield Avenue
Champaign, IL 61820

or, via electronic mail at:

softdev@ncsa.uiuc.edu
softdev@ncsavms.bitnet
In the functions presented in the following sections, substitute the name of the data array to be manipulated for the variable *q*.

Trigonometric Functions

NCSA DataScope supports a variety of standard math functions, including all of the standard trigonometric functions in radians and in degrees.

The following trigonometric angles are in radians.

sin(q)	asin(q)	sinh(q)
cos(q)	acos(q)	cosh(q)
tan(q)	atan(q)	tanh(q)
atan2(q,p)		

The following trigonometric angles are in degrees.

dsin(q)	dasin(q)	dsinh(q)
dcos(q)	dacos(q)	dcosh(q)
dtan(q)	datan(q)	dtanh(q)
datan2(q,p)		

So that you can change the data itself, the following conversions are provided.

dtor(q)	degrees to radians
rtod(q)	radians to degrees

Mathematical Functions

Similar to a good scientific calculator, the following math functions round out the set of simple one-to-one functions.

log(q)	natural logarithm
exp(q)	exponential function
log10(q)	base 10 logarithm
pow(q,p)	take q to the power p
sqrt(q)	square root
abs(q)	absolute value

The following functions return a single value computed from the entire array of values, and report the value in the notebook window.

mean(q)	arithmetic mean
sdev(q)	standard deviation
min(q)	minimum data value
max(q)	maximum data value

Data Manipulation Functions

Because NCSA DataScope works with arrays instead of individual numbers, there are several functions which may come in handy for more complex calculations.

pts(q)	number of points in array q
cols(q)	number of columns in array q
rows(q)	number of rows in array q
colrange(q)	range of column scale values
rowrange(q)	range of row scale values
colmean(q)	mean of distance between columns
rowmean(q)	mean of distance between rows
colsdev(q)	standard deviation of distance between columns
rowsdev(q)	standard deviation of distance between rows

Transformation Functions

Frequently, you may write out arrays in FORTRAN (column-major) array order, but NCSA DataScope reads the data in C (row-major) array order. To restore your desired orientation of the data, NCSA DataScope provides a transpose function, which exchanges the row and column labels and flips the data according to the different interpretation. The operation is reversible by calling transpose again.

transpose(q)

The shift functions are very useful in calculations to change from one row to another and from one column to another. During the operation, the columns and rows at the trailing edge are filled in with duplicates of the edge column or row.

shl(q)	shift data to the left one column
shr(q)	shift data to the right one column
shu(q)	shift data up one row
shd(q)	shift data down one row

For example, $\Delta X = X_{i+1} - X_i$ can be calculated by taking

deltax = shr(q) – q

NOTE: Beware of unintended effects near the border of your resulting dataset.

Built-In Kernels

NCSA DataScope can perform a variety of kernel convolution operations. These operations use the nearest neighbors of each data value to produce a new dataset. Each neighbor is multiplied with a constant (part of the kernel) and added into the kernel sum. The final sum is divided by a predetermined value, or left as the aggregate sum.

For all of the convolution functions, the border values are not calculated until the last stage. They are copied from the nearest valid data point on the edge.

In the first set of convolutions, this method is used to determine an approximation of the derivative at a point. It is only accurate to the extent that the surface lacks sudden transitions, and it only uses three adjacent values to come up with the approximation.

ddx(q)	dq/dx – difference from left to right
ddy(q)	dq/dy – difference from top to bottom
d2dx(q)	d2q/dx2 – second derivative from left to right
d2dy(q)	d2q/dy2 – second derivative from top to bottom

The following Laplacian approximations are only valid if you have a constant spacing of rows and columns.

lap(q)	5 point laplacian
lap5(q)	same as lap(q)
lap9(q)	9 point laplacian

Generic kernel operations are implemented with the kernel function:

kernel(q,k)	generic kernel

where *q* is the data array to be operated on and *k* is the kernel.

The generic kernel array must be in a special form. Starting with a 3 x 3, 5 x 5, or 7 x 7 kernel, you must add one row at the top which contains the division constant, so a 3 x 3 kernel becomes 4 x 3, a 5 x 5 kernel becomes 6 x 5, and a 7 x 7 kernel becomes 8 x 7. The division constant is located at the upper-left corner of the kernel and the rest of the first row is ignored.

Kernel convolutions are described in this chapter's section entitled "Kernels for Built-In Functions."

Nesting Functions

NCSA DataScope allows you to nest functions. This allows you to create equations as they are to be in final form, without going through all of the intermediate steps. For example, to normalize the range of values in array B to a logarithmic scale, approximately 0 to 1, the following formulas could be used in succession or combined together.

Separate:

```
X = B - min(B)
Y = X / (max(B) - min(B))
Z = .1 + Y
A = log(Z)
```

Combined:

```
A = Log (.1 + (B—min(B)/(max(B)—min(B)))
```

Kernel Convolution Functions

A 3 x 3 computational kernel has the layout shown in Figure 4.3, where the value to be computed is in the center. The individual elements of the kernel made up coefficients in the equation used to find a new center value. The coefficients are multiplied with the data values in the array and added together. This kernel operation is repeated once for each member of the data array.

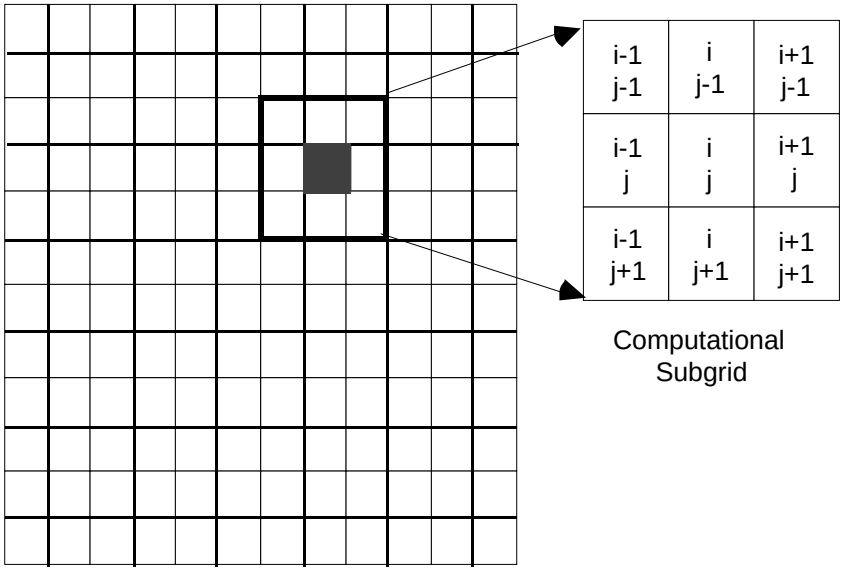
Figure 4.3 Layout of 3x3 Kernel

i-1 j-1	i j-1	i+1 j-1
i-1 j	i j	i+1 j
i-1 j+1	i j+1	i+1 j+1

As shown in Figure 4.4, the shaded square marks the current array element to be computed. The surrounding 3 x 3 grid of nine elements is used to compute the value in the middle for the result array. The surrounding 3 x 3 grid of numbers is used for each point in the resulting array, along with the 3 x 3 kernel of coefficients, to compute the results with the following formula. If array K contains the nine kernel coefficients, and array P contains the computational subgrid diagrammed in Figure 4.4, the resulting value for the center, $f_{i,j}$ is given by:

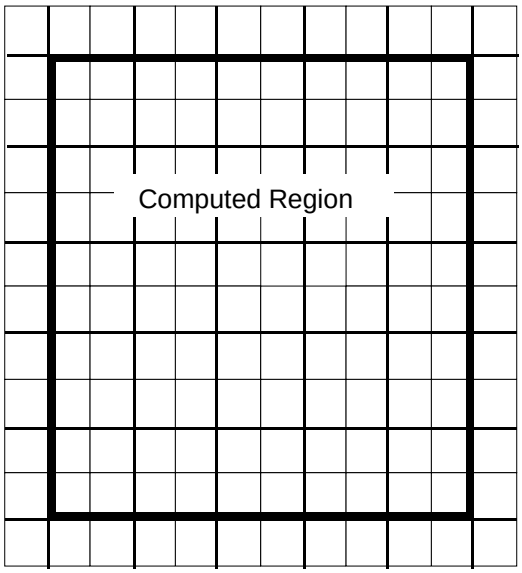
$$f_{i,j} = \S\DO2\UP2(\sum_{a=i-1}^{i+1}) \S\DO2\UP2(\sum_{b=j-1}^{j+1}) P_{a,b} K_{a,b}$$

Figure 4.4 **3x3 Kernel in Array Grid**



Calculations are done once for each point within the computed region, shown in Figure 4.5. Because the kernel computations require neighbors on each side in all directions, the computed region extends to one element from the edge. At the end of the kernel operation, the edge values are filled in by copying them from their nearest neighbors.

Figure 4.5 **Computed Region**



When using the generic kernel function, you provide the names of two arrays, one to be used for data input, and the other containing the kernel coefficients. The computation proceeds across the rows of the dataset and down, evaluating the kernel at each point to produce an answer for the resulting array. The next section describes how to set up the kernel array.

Kernel Array Construction

For example, to yield the 3 x 3 kernel shown in Figure 4.6, you could input either of the kernels shown in Figure 4.7, where the leftmost value of the first row (3 and 1, respectively) is the division constant for the array formed by the remaining rows and the other values of the first row are irrelevant. (The symbol "Ø" means "don't care"; any value may be there since these values are ignored.)

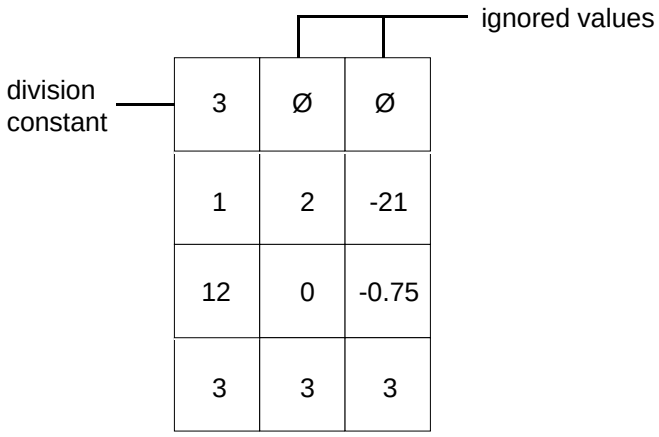
Figure 4.6 Desired 3 x 3 Kernel

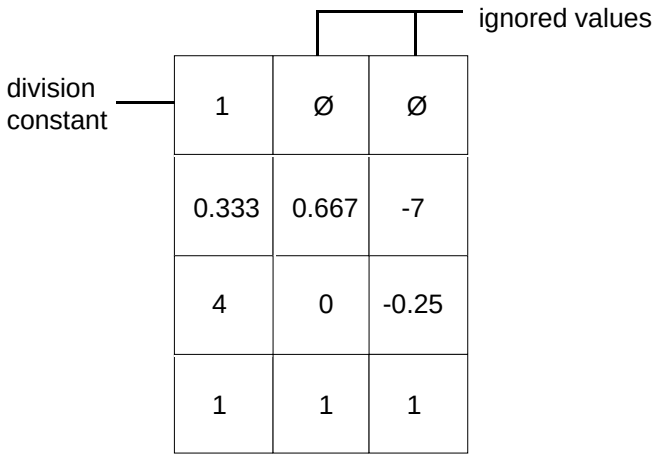
rows

1/3	2/3	-7
4	0	-1/4
1	1	1

columns

Figure 4.7 Sample Input Kernels





Generic Kernel Examples

For example, to perform a differencing equation to approximate the x derivative of an array which has equally spaced columns, one unit apart:

1. Choose Open from the File menu or press ⌘-O; and select and open the vortex.hdf file to load the vertical_vel field into NCSA DataScope.

2. Select See Notebook from the Numbers menu or press ⌘-N. A notebook window appears.

3. Enter the following array into the notebook window.

```
 2    0    0
 0    0    0
-1    0    1
 0    0    0
```

4. Select the array.

5. Choose Copy from the Edit menu or press ⌘-C. The array is copied onto the Clipboard.

6. Select a non-notebook window.

7. Choose Paste from the Edit menu or press ⌘-V. This creates a new text window.

8. Name the window d_kernel using the Attributes dialog box (see "Specifying Text Window Characteristics" in Chapter 3). You now have a usable kernel.

9. Enter and select the following formula in the notebook window.

   diffx = kernel(vertical_vel, d_kernel)

10. Choose Calculate From Notes from the Numbers menu or press ⌘-R. The command applies your 3x3 convolution to the data and produces a new dataset the same size as vertical_vel.

Look at the image to verify that each data point consists of the value to its right minus the value to its left divided by 2.

Try using the kernels shown in Figures 4.8 through 4.10 for experimentation. The kernel shown in Figure 4.8 smooths the data, averaging the center value with its neighbors, reducing any spikes and sharp transitions. The left kernel in Figure 4.9 brings out horizontal lines in the data; the right kernel brings out vertical lines. Finally, the kernel in Figure 4.10 sharpens edges in the data.

Figure 4.8 Kernel That Smooths Data

1	0	1
0	1	0
1	0	1

Figure 4.9 Kernels That Detect Horizontal and Vertical Lines of Data

1	1	1
0	0	0
-1	-1	-1

1	0	-1
1	0	-1
1	0	-1

Figure 4.10 Kernel That Sharpens Edges in Data

1	-1	1
-1	1	-1
1	-1	1

Kernels for Built-In Functions

This section presents the coefficient matrices for the built-in kernel functions (Figures 4.11 through 4.16). Certain constants are used throughout:

- the width or mesh spacing between the rows and columns—Δx and Δy:

  $\Delta x = (x_{max} - x_{min}) / (n_{cols} - 1)$
  $\Delta y = (y_{max} - y_{min}) / (n_{rows} - 1)$

  NOTE: These constants are valid only when the columns are evenly spaced.

If the width or mesh spacing between the rows and columns are *not* evenly spaced, then:

- the row and/or column differences for a point—h_i and h_j:

  $h_i = x_{i+1} - x_i$
  $h_j = y_{j+1} - y_j$

  where x_i is the column scale value at column i and y_j is the row scale value at row j.

Figure 4.11 ddx Kernel

A	B	C

$A = -h_i / [h_{i-1} (h_i + h_{i-1})]$
$B = -(A + C)$
$C = h_{i-1} / [h_i (h_i + h_{i-1})]$

$f_{i,j} = A*P_{i-1,j} + B*P_{i,j} + C*P_{i+1,j}$

Figure 4.12 ddy Kernel

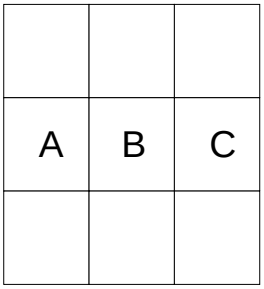
	A	
	B	
	C	

$A = -h_j / [h_{j-1} (h_j + h_{j-1})]$
$B = -(A + C)$

$$C = h_{j-1} / [\, h_j \,(h_j + h_{j-1})]$$

$$f_{i,j} = A*P_{i,j-1} + B*P_{i,j} + C*P_{i,j+1}$$

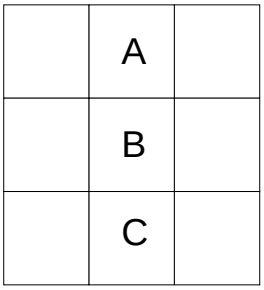
Figure 4.13 d2dx Kernel



$A = 2 / [h_{i-1} (h_i + h_{i-1})]$
$B = – (A + C)$
$C = 2/ [h_i (h_i + h_{i-1})]$

$f_{i,j} = A*P_{i-1,j} + B*P_{i,j} + C*P_{i+1,j}$

Figure 4.14 d2dy Kernel

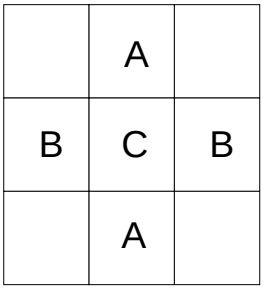


$A = 2 / [h_{j-1} (h_j + h_{j-1})]$
$B = – (A + C)$
$C = 2 / [h_j (h_j + h_{j-1})]$

$f_{i,j} = A*P_{i,j-1} + B*P_{i,j} + C*P_{i,j+1}$

Because of the use of Δx and Δy in the Laplacian approximations, they are only valid for evenly spaced rows and columns.

Figure 4.15 lap and lap5 Kernel



$A = 1/(\Delta y)^2$
$B = 1/(\Delta x)^2$
$C = –2 (A + B)$

$$f_{i,j} = A*(P_{i,j-1} + P_{i,j+1}) + B*(P_{i-1,j} + P_{i+1,j}) + C*P_{i,j}$$

Figure 4.16 **lap9 Kernel**

A	B	A
B	C	B
A	B	A

$$\Delta h = (\Delta x + \Delta y)/2$$
$$A = 1/[4\ (\Delta h)^2]$$
$$B = 2A$$
$$C = -12A$$

$$f_{i,j} = A*(P_{i-1,j-1} + P_{i-1,j+1} + P_{i+1,j-1} + P_{i+1,j+1}) + B*(P_{i,j-1} + P_{i,j+1} + P_{i-1,j} + P_{i+1,j}) + C*P_{i,j}$$

External Libraries of Functions

NCSA DataScope comes with a sample external library called `DSlibrary`. The source code to this library, which can be compiled under MPW3.0 C, is included. The library contains a few useful functions that may be used in the same manner that DataScope's built-in functions are used from the notebook. This section explains how to write and use your own library of functions.

Using the External Library Functions

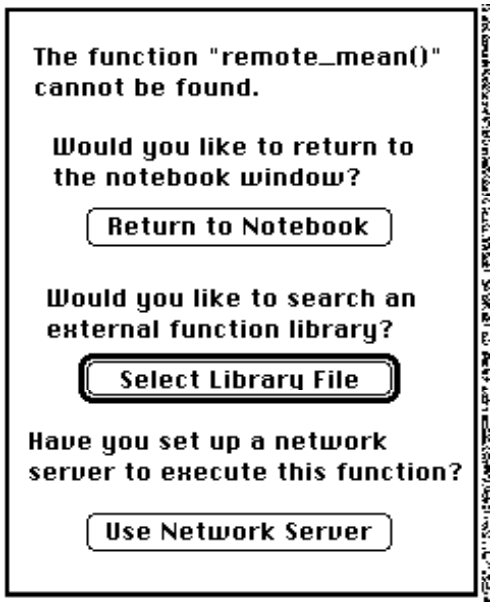
The external DataScope library `DSlibrary` contains five functions that you can use in the same manner that you use DataScope's built-in functions. The functions are listed in Table 4.1 (`P` and `Q` denote input arrays or constants).

Table 4.1 DSlibrary Functions

Function	Description
`genconst(P,Q)`	Given an array and a constant, generate an array where the output dimensions are the same as the input dimensions and where each element of the array has the value of the constant. If `P` is a 256 x 128 array and `Q` is the constant 5.2, then the output is an array of 256 x 128 where every element has the value 5.2.
`lesser(P,Q)`	Given 2 arrays or 2 constants or a constant and an array, the output is the lesser value.
`greater(P,Q)`	Given 2 arrays or 2 constants or a constant and an array, the output is the greater value.
`rowarray(P)`	Given an array, the values of the output are the corresponding column scale value for every element in the array.
`colarray(P)`	Given an array, the values of the output are the corresponding row scale value for every element in the array.

Anytime NCSA DataScope cannot find a function name in its internal list, the dialog box in Figure 4.17 appears on the screen to help you choose an external library. If you misspelled the function name, click the Return to Notebook button; otherwise click Select Library File and proceed to select and open the library from disk. Once you open a library, you can use any of the functions within it. All the functions remain available until you exit NCSA DataScope.

Figure 4.17 **External Function Library Dialog Box**



Creating an External Function

To create your own external function, include the structure definition code, shown in Figure 4.19. This is the format for the data which is passed to and from the external function. First, though, define a function `addon()` (Figure 4.18), or any suitable name, which accepts three parameters, two operands and a result. This code segment must be the first executable code in the C file. In your function, take information from the left and/or right parameters and place the resulting information into the structure provided for the answer.

Figure 4.18 **Defining a Function Addon**

```
long addon(l,r,a)
  scope_array *l,*r,*a;
{

  /* contents of function */
}
```

Compile this function in MPW3.0 C with the following compile line:

```
C myfile.c -o myfile.o
```

Next , enter the link line, which has several parts that are specifically set up for creating code segments that NCSA DataScope can execute.

```
link -m addon -rt DSfn=1000 -sn Main=myfunction ∂
myfile.o -c 'NCSf' -t 'DSff' -o myextern.lib
```

Within the link line:

- The parameter *-m addon* indicates that the code segment you are calling is named addon (use all uppercase letters for Pascal calling conventions).
- The *-rt DSfn=1000* parameter sets up the resource information for the code segment. DSfn is the special resource type which NCSA DataScope looks for when it attempts to open an external library. 1000 is the resource ID number assigned to this function. If you use more than one function in one file, change this number for each function; for example, 1001, 1002, 1003....
- The *-sn Main=myfunction* parameter assigns a name to the code resource. In the notebook window, this function is called with the name *myfunction*. You do not have to have this name anywhere in the source code file in order to call it from the notebook.
- The file type and creator settings, *-c 'NCSf'* and *-t 'DSff'* set the icon for the file and allow NCSA DataScope to find the file. Only files with this type can be opened as external function libraries.

The rest of this link line is comprised of standard parameters to the link command.

- *myfile.o* is the file to be linked (from the C compile line).
- ∂ indicates a continuation from one line to another in the MPW shell; it is not part of the link line itself.
- *-o myextern.lib* chooses the filename for the output.

Working examples of the external functions for DSlibrary are included on the distribution disk in the folder "externs".

Figure 4.19 Structure Definition Code

```
/*
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Data structure for external functions.

            The field "kind" determines whether the array contains a valid
            cval or a valid set of ncols,nrows,rows,cols,vals.

            External functions are called as:
            your_fn(lft,rgt,answer)
            scope_array *lft,                  left parameter
                        *rgt,                  right parameter
                        *answer;               place to put the answer

            Answer will always contain pre-allocated space for an array of
            resulting values, including the rows and cols arrays. You may
            change any value (and should) in the rows,cols and vals arrays.
            Do not change any values in the lft or rgt storage.

            If your routine returns only a constant, set kind == DS_CONSTANT
            and put the answer in cval.

            DON'T allocate anything you don't free yourself.
            FREE everything you allocate.
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*/

#define DS_ERROR -1
#define DS_CONSTANT 0
#define DS_ARRAY 1

typedef struct
                    {float
                          cval,  /* constant value when we are carrying a constant */
                          *rows, /* row labels, scale values:  count = ncols */
                          *cols, /* col labels, scale values:  count = nrows */
                          *vals; /* data values in the array, if there is an
                          array        size = ncols*nrows           */

                     int   ncols,
                          nrows; /* dimensions of the array */

                     char
                          kind;         /* ERROR, CONSTANT, ARRAY */
                    }

                    scope_array;
```

Creating Your Own Library Functions

Figure 4.20 shows the source code for the `genconst` function in the DataScope external function library. Note that:

- The header file is included
- The passed parameters l, r, and a (corresponding to the left input, the right input, and the answer output, respectively:
  `a = genconst(l,r)` in DataScope's notebook) are declared.
- Appropriate error checking is performed (To take the dimensions from one input array and

assign all of the

elements of the output to have a constant value, the two inputs must be an array and a constant. You may enter the two inputs in any order)

Once these preliminaries are covered, the code that actually does the work is simple.

Figure 4.20 Source Code for Function Genconst

```
#include "DScope.h"
/*
              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              genconst      Given an array and a constant, generate
                            an array where the output dimensions are
                            the same as the input dimensions and
                            where each element of the array has the
                            value of the constant
              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*/
long addon(l,r,a)
              scope_array         *l,*r,*a;
{
              register float      *p,*q;

              if (!l || !r)
                 {a->kind = DS_ERROR;
                     return(0);
                 }

              if (l->kind == DS_CONSTANT && r->kind == DS_CONSTANT)
                 {a->kind = DS_ERROR;
                     return(0);
                 }

              if (l->kind == DS_ARRAY && r->kind == DS_ARRAY)
                 {a->kind = DS_ERROR;
                     return(0);
                 }

              p = a->vals;
              q = p + (long)(a->ncols * a->nrows - 1);

              if (l->kind == DS_CONSTANT)
                 {while (p < q)   *p++ = l->cval;
                     *p = l->cval;
                  return (0);
                 }

              while (p < q) *p++ = r->cval;
              *p = r->cval;
              return (0);
}
```

Figure 4.21 shows the Makefile used to create DataScope's `DSlibrary`. Symbolic parameters make the inclusion of addition functions a straightforward procedure. Note that the link options to create function require MPW's `CRuntime.o` file (needed to perform the unsigned long multiplication): It is your responsibility to determine which compile and link options are appropriate for a given program.

Figure 4.21 **Makefile Used to Create DSlibrary**

```
#######################################
#             make external library for DataScope
#######################################
LOPT          =      -m     addon         ∂
                            -c     'NCSf'        ∂
                            -t     'DSff'        ∂
                            rm.o                 ∂
                            -o     DSlibrary
#
COPT          =      -o     rm.o
#
FILE1         =      colarray.c
LOPT1         =      -rt    DSfn=1000     -sn Main=colarray
#
FILE2         =      rowarray.c
LOPT2         =      -rt    DSfn=1001     -sn Main=rowarray
#
FILE3         =      lesser.c
LOPT3         =      -rt    DSfn=1002     -sn Main=lesser            ∂
                            "{CLibraries}"CRuntime.o
#
FILE4         =      greater.c
LOPT4         =      -rt    DSfn=1003     -sn Main=greater    ∂
                            "{CLibraries}"CRuntime.o
#
FILE5         =      genconst.c
LOPT5         =      -rt    DSfn=1004     -sn Main=genconst   ∂
                            "{CLibraries}"CRuntime.o
#
FILES         =      {FILE1}              ∂
                            {FILE2}              ∂
                            {FILE3}              ∂
                            {FILE4}              ∂
                            {FILE5}
#######################################
DSlibrary     ƒ      {FILE1}       ∂
                                   {FILE2}       ∂
                                   {FILE3}       ∂
                                   {FILE4}       ∂
                                   {FILE5}
              C             {FILE1}              {COPT}
              Link   {LOPT}        {LOPT1}
              C             {FILE2}              {COPT}
              Link   {LOPT}        {LOPT2}
              C             {FILE3}              {COPT}
              Link   {LOPT}        {LOPT3}
              C             {FILE4}              {COPT}
              Link   {LOPT}        {LOPT4}
              C             {FILE5}              {COPT}
              Link   {LOPT}        {LOPT5}
#
```