URL: http://cxc.harvard.edu/ciao3.4/dmtcalc_expressions.html
Last modified: December 2006

AHELP for CIAO 3.4

syntax

Context: tools

Jump to: Description EXAMPLES CHANGES in CIAO 3.0 WHITESPACE IN EXPRESSIONS Bugs See Also

Synopsis

Syntax used for writing mathematical operations in dmtcalc, dmimgcalc, and dmgti

Description

The dmtcalc, dmimgcalc, and dmgti tools create new files from the contents of existing ones − e.g. adding two images together or dividing one column by another. This document describes the syntax that is understood by these tools. Since the tools behave differently − for instance dmtcalc and dmgti work on tables whilst dmimgcalc work on images − not all the functionality described below is available to each program.

Supported functionality:

- Creating new columns and over−writing existing ones (dmtcalc only).
- Full support for the Data Model's vectors and arrays.
- Operators and Functions: +, −, /, *, % (modulo), exponentiation (N**M or N^M), <, >, <=, >=, ==, !=, !, &&, ||, ~= (check for a substring), cos, sin, tan, acos, asin, atan, cosh, sinh, tanh, exp, log, ln, sqrt, fabs, the constant pi, random numbers, NaN values, and the column row number.
- Scientific notation: real numbers can be specified using eEdD notation (eg 1.0e−3, 6d4).
- Access to the values of keywords in the headers of the input files.
- String constants can be included if surrounded in quotes.
- Conditional expressions: ?:, if/then, if/then/else.
- Creation and use of temporary values (dmtcalc only).
- Smoothing of columns (dmtcalc and dmgti).
- Bit checking and assignment (dmtcalc and dmgti).

Using column or image data from the input files

To refer to a column in dmtcalc or dmgti just use its name (which can be found by using dmlist with the cols option or with the graphical file browser prism); the case of the column name is not important. For dmimgcalc, the input images are referred to as img1 through imgn (where n is the number of images in the input stack); again case is not important.

In dmtcalc the values in existing columns can be changed by saying

```
colname=<expression>
```

where the format of "<expression>" is described below. The datatype and dimensionality of the column can not be changed unless it is not used in any other expression (i.e. no other column is defined in terms of it).

Creating new columns or images

A new column in dmtcalc is created by saying

```
newname=<expression>
```

− i.e. it is similar to changing the values of an existing column except that the data type and dimensionality of the column can be set. The number of rows of the output column is set by the number of rows in the table and the dimensionality of the column by the result of "<expression>". The above expression would create, at the end of the columns, a new column called NEWNAME in the output file; the output name is converted to upper case however it is written in the expression. It is similar for dmgti, except that the column will only be created if the mtlfile is created (this is optional).

For dmimgcalc the output image is referred to as "imgout" (case is again unimportant); the name of the file is taken from the outfile parameter.

As a simple example: the expression

```
energy=((float)pha*0.1)
```

would − if given to dmtcalc − change the energy column of an event file to values that are ten percent of those of the PHA column. Since the pha column is stored as an integer, "(float)" is used to convert (typecast) the values to real numbers for the calculation (note that this equation is not valid for Chandra data and is just used as an example). Since the energy column has been changed, it would be written out with its name in upper case, whatever the value used in the input file. Any column that is changed is also moved to the end of the column list of that table. To change the order of the columns in a table file you can use the dmcopy tool, using the "cols" option with the desired new column order.

Similarly,

```
imgout=(img1/sqrt(fabs(img2)))
```

tells dmimgcalc to create the output image by dividing the first input image by the square root of the absolute values of the second image. Here the case is not important since the names "imgout", "img1", and "img2" are a means of identifying datasets rather than file (or column) names.

Using () to surround expressions

In the examples shown in this document, example expressions will generally be surrounded by ()'s. This is to ensure that they evaluate correctly, and so that white space is not an issue (see the "WHITESPACE IN EXPRESSIONS" section at the end of the document).

Using header values

Header values can be used in the expressions. For dmtcalc and dmgti you just use the name of the header keyword (case insensitive); for example the dmtcalc expression

```
dtime=(time−TSTART)
```

means that the column DTIME should be created by subtracting the value of the TSTART keyword from each row of the TIME column.

For dmimgcalc you also need to specify which file the keyword is to be taken from: this is done by appending the keyword name to "img<n>_", where <n> is the number of the file. So

```
imgout=(img2+img1*img2_exptime/img1_exptime)
```

creates an output image equal to the sum of the two images after multiplying the first image by the ratio of the EXPTIME keyword values (image 2's exposure time divided by image 1's exposure time).

Temporary columns

In dmtcalc you can create a temporary column to simplify expressions but which will not be written to the output file. They can be thought of as temporary variables used in the expression. A temporary column is created as a new column would be except that the name must start with the "." character. This means that the following two statements are equivalent for dmtcalc:

```
eval=((pha−15.0)/12.34)
```

and

```
.dpha=(pha−15.0)
```

```
eval=(.dpha/12.34)
```

Numbers and Strings

Scientific notation can be used for numbers, so

```
newcol=(oldcol*0.01)
```

and

```
newcol=(oldcol*1.0e−2)
```

have the same meaning. The parser accepts any of e, E, d, or D for the label separating the mantissaa from the exponent, so you can say 1.0e−2 or 1d−2. String values can be used in dmgti and dmtcalc − for instance if you want to filter or manipulate a string column − by surrounding the value in matching quotes, either single (') or double ("). For boolean values, 1 and T are used to indicate TRUE whilst 0 and F are used to indicate FALSE; T and F should not be quoted in this case.

Operators

Common mathematical operators, such as multiplication and addition, are allowed between scalars and arrays (whether one or more dimensions). When combining a scalar with an array then the scalar value is applied to each element of the array, which is why

```
newcol=(oldcol*0.01)
```

creates a column where each row is one percent of the value of the corresponding row in the array represented by "oldcol".

The supported operators and functions are listed in the following tables. Note that columns containing bit values − such as the STATUS column of a Chandra event file − have their syntax which is described later in the document.

Supported operations

Operator	Description
+	Addition.
−	Subtraction.

*	Multiplication.
/	Division.
%	Modulo: so "7 % 3" is 1.
** or ^	Exponentiation: so 7**3 equals 7^3 (and is 343).

Supported boolean operations, and if they work on numbers or strings.

Operator	Example	Description	Numbers	Strings
<	rate < 2.0	Less than.	Yes	No
>	rate > 2.0	Greater than.	Yes	No
<=	rate <= 2.0	Less than or equal to.	Yes	No
>=	rate >= 2.0	Greater than or equal.	Yes	No
==	component=='circle'	Are the two values equal.	Yes	Yes
!=	component!='circle'	Are the two values different.	Yes	Yes
~=	component~='cir'	Is the right−hand side string a sub−string of the left−hand side value?	No	Yes

The result of a boolean operation such as

```
rate>2.0
```

is a boolean. This can be used to set the value of a row or pixel, combined using one of !, && or \|\| as described in the following table, or used in a conditional statement.

Operators that work on booleans

Operator	Example	Description
!	!(rate>5.0)	Negate the value (so TRUE to FALSE and vice−versa)
&&	(rate>2)&&(rate<4)	Logically "and" the two boolean values.
||	(rate<0)||(rate>5.0)	Logically "or" the two boolean values.

The result of using !, &&, or || is another boolean expression.

Supported Functions

Function	Description
cos	Cosine. Argument is in radians.
sin	Sine. Argument is in radians.
tan	Tan. Argument is in radians.
acos	Inverse cosine
asin	Inverse sine
atan	Inverse tan
cosh	Hyperbolic cosine
sinh	Hyperbolic sine
tanh	Hyperbolic tan
exp	Exponential
log	Logarithm (base 10)
ln	Logarithm (base e)
sqrt	Square root
fabs	Return the absolute value

The functions listed above must be given in lower case. They can operate on scalars or arrays, so if given a column (i.e. array) of ten elements, cos() will return a ten−element array where each element is the cosine of the corresponding element in the original array. The functions work with double precision numbers; input values will be typecast (ie converted) to double format if necessary and the result typecast back to the input data type.

Column Smoothing

In dmtcalc and dmgti, numeric columns can be smoothed before the values are used. This smoothing is a simple average of an odd number of rows − centered on the current row − and is indicated by appending ":<n>" to the column name, where "<n>" is an integer. The value will be increased by one if it is an even value and is taken to be 1 (so no smoothing is applied) if no value after the column is given. The smoothing at the start and end of the table will be restricted to the available number of rows (so the first row will only be averaged over "n−(n−1)/2" rows).

As an example,

```
srate=(rate:5−0.123)
```

creates a new column called SRATE which equals the smoothed rate column, averaged over 5 rows, minus 0.123.

Once a column has been smoothed, those smoothed values will be used whenever that column is referenced − whether or not a different smoothing factor is specified. However, different columns can be smoothed by different values.

Special symbols

The character "#" is used to identify a special symbol for the parser. Currently there are five such symbols:

- #row
- #pi
- #nan
- #rand
- #trand

Whenever #row is used it will be replaced by the current row number, #pi will be replaced by the value of PI, #nan is used to indicate the "Not a Number" value for floating−point values, and the #rand and #trand symbols create random numbers, as described in the next section. Note that there is currently no way to indicate the NULL value for an integer column or array.

Random numbers

Both #rand and #trand will return a random number between 0 and 1 (inclusive). The difference between them is that #trand causes the seed for the random number generator to be taken from the system clock whereas #rand uses 1 as the seed and #rand(<N>) uses the value of <N> (which must be an integer > 0) as the seed value.

The seeding of the random number generator occurs before the expressions are evaluated (i.e. only once) and only the first occurrence of #rand or #trand is used to determine the seed value.

Conditional Expressions

Various conditional expressions are supported. The following will allow some degree of variability in what gets evaluated:

- if(expression)then(expression)else(expression)
- if(expression)then(expression)
- (expression)?(expression):(expression)

As an example, the following expression would − if given to dmtcalc − cause all rows of the psfratio column which equalled NaN (i.e. Not A Number) to be replaced by 0, whilst all other row values are copied across unchanged:

```
if(psfratio==#nan)then(psfratio=0)
```

Multiple statements − including setting a new column − can be included within the if/then and if/then/else constructs by separating the expressions with semi−colons. So, the following statement has the same result on the psfratio column but also creates a new column − FLAGCOL − whose value is one if the psfratio was NaN (and hence has been set to 0) and zero otherwise:

```
if(psfratio==#nan)then(psfratio=0;flagcol=1)else(flagcol=0)
```

The construct "(expression)?(expression):(expression)" is a shortcut for the "if/then/else" construct except that the expression term can not contain multiple statements. As an example,

```
(psfratio==#nan)?(flagcol=1):(flagcol=0)
```

creates the FLAGCOL column with values as above but does not change the psfratio column.

Converting numeric values from one format to another

Whilst the row or pixel values are converted automatically to sensible form − so the result of "10.0*pha" will be in floating−point if the pha values are integers − it is sometimes necessary to explicitly specify that a conversion between data formats is needed. This is known as "casting" or "typecasting", and is indicated by preceeding the value to be converted − be it a column name or the result of an expression − by the name of the new data type surrounded in ()'s. So

```
outimg=((double)img1)
```

means that the output image will be a copy of the input image but converted to double format.

List of supported casts

Cast Name	Data Type
bool	boolean
ushort	unsigned short
short	short
byte	unsigned char
uchar	unsigned char
ulong	unsigned long
long	long

float	float
double	double

The use of "(int)" is also allowed but it is not guaranteed to work correctly so you should use one of the types above − such as "(long)" or "(short)" − instead.

Case sensitivity

The mathematical functions such as cos and exp must be written in lower case. Case is not important when referring to column names, image labels (e.g. "img2") in dmimgcalc, or header keywords. However, note that the names of new and updated columns will be written out in upper case by dmtcalc.

Creating columns

Previous examples have created a new column by assigning it the results of a calculation, such as "(rate−0.123)". It is also possible to define a column − in dmtcalc − by giving its data type and dimensionality and then assigning values to it. The name of the data type should match one of the casts listed above and the array dimensions are indicated by surrounding each dimension size in []'s. If no dimensions are given then it is assumed that it is a scalar column (i.e. there is only one element per row).

The expression

```
shortcol=short
```

will create a column called SHORTCOL which contains Int2 values, all set to zero, and there will only be one element per row. The expression

```
shortcol=short[2]
```

also creates a column called SHORTCOL which contains Int2 values, but this time there are two values per row (so the column is actually a two−dimensional array).

Columns of string values can be created by saying

```
strcol=(text 32)
```

which creates a column called STRCOL which contains strings of length 32 characters. Bit columns follow a similar form in that

```
bitcol=(bit 6)
```

creates a column called BITCOL which has 6 bits per row.

Numeric columns are created containing zeroes, bit columns are created with all bits unset, and string columns are created with the row values set to "".

Values can be assigned to columns after they have been created by referencing the column name. So

```
shortcol=short,shortcol=−3
```

creates a column of Int2 values and then sets each row to the value −3, whereas

```
shortcol=short,shortcol=energy/100.0
```

will set SHORTCOL to one percent of the energy column (with an implicit typecast to Int2 format).

Multi−dimensional arrays can be set by surrounding a set of numbers with {}'s. The expression

```
shortcol=short[2],shortcol={−3,4}
```

sets each row of SHORTCOL to the array [−3,4]. The contents within the {} can be any valid expression, although it is suggested that temporary variables − ie names beginning with a "." − are used to keep each line to a manageable length.

In fact it is not always necessary to define the array dimensionality before use, depending on what data type and dimensionality you want the new column to have. The expression

```
arrcol={-3,4}
```

will creates a column of integers (of type long) where each row contains two elements (−3 and 4), whereas

```
arrcol={-3,4.0}
```

creates a similar column except that the data type is double rather than long.

Setting multi−dimensional arrays

The previous technique only works for columns whose rows are either scalars or one−dimensional arrays. For rows containing arrays of higher dimension, the size must be explicitly defined:

```
nphas=short[3][3]
```

creates a column called NPHAS which − in each row − contains a 3x3 array of Int2 values. Such arrays can be set using {}'s by treating each row as a one−dimensional array whose number of elements equals the number of elements in the array (here 9). So the following expression is valid

```
  nphas=short[3][3]
  nphas=phas + { −3,4,0, 1,2,7, −8,4,0 }
```

It assumes that the column called phas in the input file also contains 3x3 elements per row and is given as an example (it is extremely unlikely that you would want to change the PHAS column of your level 1 Chandra ACIS event file in this way!).

Accessing individual elements

Individual elements of a column can be accessed using "[]" to indicate the element location, so

```
ncol={1,4,3,2}
```

and

```
  ncol=short[4]
  ncol[0]=1
  ncol[1]=4
  ncol[2]=3
  ncol[3]=2
```

are equal. Note that arrays are indexed counting from 0.

This is not limited to one−dimensional arrays of elements. Using the nphas/phas columns from a previous example,

```
nphas[2][1]=phas[1][0]
```

will set the (2,1) element of each row of the NPHAS column to the (1,0) element of the phas column.

Working with BIT columns

Both dmtcalc and dmgti can manipulate BIT columns, such as the STATUS column of Chandra event files. Note that the syntax does not match that used by the Data Model (as described in "ahelp dmfiltering").

Checking bit columns

To check if a particular bit is set (i.e. TRUE) or unset (i.e. FALSE), you write X<N>T or X<N>F respectively, where <N> should be set to the bit number. This number starts at 0 so "X3F" returns TRUE if the fourth bit is unset. Multiple bits can be checked by using a comma−separated list of terms, so

```
status==X1F,X3T,X18F
```

is true if the second and nineteenth bits are unset and the fourth bit is set (it ignores the values of all other bits).

To check if all bits are unset compare the column to "0" − i.e.

```
status==0
```

− or "X−1F". To check that all bits are set use "X−1T", so

```
status==X−1T
```

Setting bit columns

It is only possible to assign a single bit in a single statement. Such an assignment uses the same X<N>T and X<N>F symbols as used to check bit values. To create a column called bcol containing 8 bits and then set the second and fifth bits, you would say

```
bcol=(bit 8),bcol=X1T,bcol=X4T
```

When a bit column is created all bits are unset.

EXAMPLES

See the Examples section of "ahelp dmtcalc", "ahelp dmimgcalc", and "ahelp dmgti".

CHANGES in CIAO 3.0

dmimgcalc

The dmimgcalc tool can now combine images (more than two) using the syntax described here. Previously only addition, subtraction, multiplication, and division were permitted.

The ~= operator

The ~= operator − which returns true if the string on the left contains the string on the right of the operator − was added.

#nan

NaN values in floating−point columns or images can be referenced by using the "#nan" symbol. Note that there is currently no way to indicate the NULL value of an integer column or array.

WHITESPACE IN EXPRESSIONS

The following information is not relevant for dmtcalc when the expression is stored in an ASCII file and referenced via "@<filename>". It is valid for dmtcalc when the expression is given directly and is always relevant for dmimgcalc and dmgti (since these tools can not use a stack file).

When an expression is given on the command line − so not as a stack file − then spaces are important. This is because spaces are treated as stack separators − as discussed in "ahelp stack" − which is unlikely to be the desired behaviour here.

So, setting the expression (assuming we are using dmtcalc, but it is also valid for dmgti's userlimit parameter and dmimgcalc's operation parameter) to

```
expression="energy= (float) pha / 10.0"
```

will not work. You have to say either

```
expression="energy=(float)pha/10.0"
```

− i.e. remove all the spaces − or surround the expression in ()'s, since this stops the checks for stack separators. This means that

```
expression="energy=( (float) pha / 10.0 )"
```

is a valid expression.

An alternative solution − valid for dmtcalc only − is to write the expression in a text file ("expr.lis" say) and then set

```
expression="@expr.lis"
```

since the white space in stack files is not important.

Bugs

See the bugs page for the tool you are using or for the mathematical syntax support on the CIAO website for an up−to−date listing of known bugs.

See Also

tools
: dmgti, dmimgcalc, dmtcalc, mtl_build_gti