The XMM Selector library

January 12, 2017

Abstract

selectlib provides functionality to perform table and array processing with boolean and arithmetic expressions

1 Introduction

The selector library **selectlib** provides functionality to perform table and array processing driven by boolean and arithmetic expressions. The notion of *table* and *array* is that of the Data Model defined in the SAS Data Access Layer package **dal**. Other **dal** terms such as *table column*, *data set*, *block*, and *attribute* are likewise used throughout the following sections. Readers not familiar with these notions are therefore advised to consult the **dal** documentation prior to reading any further.

2 Description

Table and array processing in **selectlib** is controlled via user-specified boolean and arithmetic expressions. These take the form of single character strings which are composed of a variety of elements including numerical and logical constants, operators, functions, and attributes. The following operations are supported by **selectlib**:

operation	required expression	comment
table filtering	logic	can contain names of other columns (see Sect. 2.15.1)
construction of new table column	arithmetic	can contain names of other columns (see Sect. 2.15.2)
construction of new array	arithmetic	can contain symbolic references to other arrays (see Sect. 2.15.3)

The following sections detail the syntax and semantics of the expressions driving the above operations and give sample expressions to demonstrate typical usage scenarios.

2.1 Boolean operators and functions

Table filtering requires the specification of a single boolean expression which must evaluate to either true or false for all rows of the table. Rows are selected if the expression evaluates to true, otherwise they are

not selected. The notion of a valid boolean expression is identical to the one in the programming languages C/C++ and Fortran: An expression is comprised of a sequence of sub-expressions combined with the logical operators `&&` (and) and `||` (or). Sub-expressions are constructed from table column names (the equivalent of variable names in C/C++/Fortran), boolean operators, and symbolic or numeric constants or other arithmetic expressions. The following table lists the available operators which can be used in their C/C++ or Fortran form:

description	C/C++ form	Fortran form
equal	`==`	`.eq.`
not equal	`!=`	`.ne.`
less than	`<`	`.lt.`
less than or equal	`<=`	`.le.`
greater than	`>`	`.gt.`
greater than or equal	`>=`	`.ge.`
or	`||`	`.or.`
and	`&&`	`.and.`
logical negation	`!`	`.not.`
inclusion test	`in` (see Sect. 2.13)	

Operator precedences are as in C/C++ and parentheses may be used as necessary to group sub-expressions. Case is insignificant, i.e., `.and.`, `.AND.`, and `.aNd.` are all valid specifications for the logical and-operator. The C/C++ forms and Fortran forms can be mixed within a single expression.

In addition to the above operators, a limited set of boolean functions is available as well. These take a number of arguments and return either true or false. The following table lists the available boolean functions:

function name	meaning	comment
`ifthenelse(expr1, expr2, expr3)`	if `expr1` is true value of construct is value of `expr2`, otherwise value of `expr3`	all three arguments must be boolean expressions
`near(val1,val2,tol)`	true if $\frac{\|\texttt{val1}-\texttt{val2}\|}{\|\texttt{val1}\|} <= \texttt{tol}$	
`point(...)`, `line(...)`, ...	point-in-figure test	see Sect. 2.10
`gti(...)`, `mask(...)`, `region(...)`	value-in-GTI/point in region tests	see Sect. 2.12
`selected()` or `selected`	selection status test	see Sect. 2.14

2.2 Arithmetic constants and identifiers

Boolean and arithmetic expressions may contain integer and floating point constants which can be specified as in C/C++ or Fortran, e.g. 1.234E-12, -23, .111222, 1e2. The value of an integer constant can be given in binary, octal, or hexadecimal by prepending `b`, `o`, and `h` or `0x` respectively, i.e., 32482 = b111111011100010 = o77342 = 0x7ee2 = h7ee2. Some frequently used constants are available as symbolic names:

name	type	value	comment
#PI	real	$\pi = 3.1415926535\ldots$	
#E	real	$e = 2.718281828\ldots$	Euler number
#RAD	real	$\pi/180 = 0.017453\ldots$	for deg $\longrightarrow$ rad conversion
#DEG	real	$180/\pi = 57.29577\ldots$	for rad $\longrightarrow$ deg conversion
#ARCSEC	real	$\pi/180/3600 = 4.8481\ldots^{-6}$	1 arcsec expressed in rad
#ARCMIN	real	$\pi/180/60 = 2.9088\ldots^{-4}$	1 arcmin expressed in rad
TRUE	boolean	true	
FALSE	boolean	false	

2.2.1 Angles

Angles can be regarded as ordinary real numbers in units of radians. In addition angles may be expressed in terms of hours, minutes, and second of arc or time (e.g. for Right Ascension, Declination):

format	comment	evaluates to	example
(+|-)xxdxxmxx[.xxx]s	degrees + minutes and seconds of arc	angle in radians	-45d23m59.9s
xxhxxmxx[.xxx]s	hours + minutes and seconds of time	angle in radians	23h59m24.1s

x stands for a single decimal digit, d, h, m, s are literal characters and square brackets denote optional items. The number of shown digits after the decimal point is not significant.

2.2.2 Times

Times can be expressed in either of the following ways:

form	comment	example
xxxx-xx-xxTxx:xx:xx[.xxx]	combined date-time string	2002-06-03T18:59:01.1
jdxxxxxx.xxx	Julian date	jd2450850.5
mjdxxxxx.xxx	Modified Julian Day	mjd50850.5
yxxxx.xxx	decimal year number	y2002.5
xxx xxx x xx:xx:xx xxxx	calendar date	Mon Jun 3 18:21:19 2002

x stands for a single decimal digit, jd, mjd, y, T are literal character strings and square brackets denote optional items. The number of shown digits after the decimal point is not significant. In an arithmetic context all given times evaluate to the number of elapsed seconds since the fixed time 1998-01-01T00:00:00 TT.

2.2.3 Identifiers

Identifiers are alphanumeric strings which can contain letters, digits and the underscore character (_). They must start with a letter — case is significant. An identifier must match a name of an existing column in the input table unless it is prefixed by a hash mark (#). In this case it must match any of the above names for symbolic constants or is otherwise interpreted as a reference to a to numerical or textual attribute in the currently processed table. The identifier #ROW has a special meaning: It stands for the current row number (starting from one) in the processed table.

Examples: #PI (the numerical value of π), #ROW (the current row number), ROWPI (a table column with name ROWPI), #ROWPI (value of the numerical attribute ROWPI).

2.3 Arithmetic operators and functions

Boolean and arithmetic expression can contain arithmetic sub-expressions consisting of operators, functions and constants. The following operators and functions are available:

description	symbol/name
arithmetic negation	`-`
addition/subtraction	`+/-`
multiplication/division	`*//`
modulus	`%`
exponentiation	`**`, `pow(x, y)`
absolute value	`abs(x)`
sine/cosine/tangent	`sin(x)/cos(x)/tan(x)`
arc sine/arc cosine	`arcsin(x)/arccos(x)`
arc tan	`arctan(x)/arctan2(x, y)` = `arctan(x/y)`
hyperbolic sine/cosine/tangent	`sinh(x)/cosh(x)/tanh(x)`
exponential	`exp(x)`
natural log	`log(x)`
common log	`log10(x)`
square root	`sqrt(x)`
integral part	`int(x)`
fractional part	`modf(x)`
smallest integral value not less than x	`ceil(x)`
largest integral value not greater than x	`floor(x)`
floating point remainder of x/y	`fmod(x, y)`

The argument of the trigonometric functions and the result of their respective inverses are angles in units of radians. For required conversions between radians and decimal degrees please avail the symbolic constants `#RAD` and `#DEG` (see Sect. 2.2). Example: `sin(ANGLE*#RAD)`, `#DEG*arcsin(VAL)`.

2.4 Vector operators and functions

A limited set of operators and functions to perform vector algebra in three-dimensional Eucledian space is available. The following table provides an overview over the available constructs:

description	symbol/name	type	comment
vector construction	`vector(x, y, z)`	vector	`x`/`y`/`z` can be arithmetic expressions
unit vector construction	`unitvector(x, y, z)`	vector	`norm(unitvector(x, y, z))==1`
construction sky vector in J2000 reference frame	`skyvector(ra, dec)`	vector	`ra`/`dec` Right Ascension/Declination in rad
unary minus	`−v`	vector	`v` vector
vector addition	`v1+v2`	vector	`v1`/`v2` vectors
vector subtraction	`v1−v2`	vector	`v1`/`v2` vectors
multiplication with scalar	`s*v`	vector	`v` vector, `s` arithmetic expression
division by scalar	`v/s`	vector	`v` vector, `s` arithmetic expression $\neq 0$
cross product	`cross(v1, v2)`	vector	`v1`/`v2` vectors
vector component	`v[i]`	scalar	`0 <= i <= 2`
vector norm	`norm(v)`	scalar	equivalent to `sqrt(v[0]**2+v[1]**2+v[2]**2)`
scalar product	`v1*v2`	scalar	equivalent to `v1[0]*v2[0]+v1[1]*v2[1]+v1[2]*v2[2]`

Example expressions involving vector algebra:

- `norm(vector(1,2,3)-4*vector(8,9,10))`
- `cross(skyvector(0,0), skyvector(#PI/2, 0))`
- `norm(vector(3,4,5)*unitvector(-2,3,4))-1`
- `5*vector(1,2,3)[0]`

2.5 Character string constants, operators and functions

Boolean subexpressions may also be formed from character string constants, identifiers that refer to text columns, and a limited set of associated operators and functions. As in C/C++ a string constant is a list of characters enclosed in double quotes (`"`). A double quote as part of the string must be preceded by a backslash. In addition, string constants may also be given as single-quoted text in which case they may contain un-escaped double quotes but no other single quote. Examples of valid string constants are: `"XMM"`, `""`, `"The double quote:\""`, `'A single-quoted string with a double quote (")'`.

The following string operators and functions are available:

description	symbol/name	`true` example expression	comment
equality	`==`, `!=`	`"X"=="X"`, `"XMM"!="XTE"`	
relational operators	`<`, `<=`, `>`, `>=`	`"a"<"b"`, `"a"<="b"`, `"b">"a"`, `"b">="a"`	lexicographical order
change case	`upper/lower`	`upper("Xmm")=="XMM"` `lower("XMM")=="xmm"`	
length	`strlen`	`strlen("XMM")==3`, `strlen("")==0`	
ASCII value	`ascii`	`ascii(" ")==32`, `ascii("Z")-ascii("A")+1==26`	
concatenation	`+`	`"Coca-"+"Cola"=="Coca-Cola"`	
range operator	`[:]`, `[:hi]`, `[lo:]`, `[lo:hi]`	`"XMM"[0:1]=="XM"` `"FREDDY"[2:]=="EDDY"`	first character has index 0; omitted lower/upper bound expands to 0/actual length-1

Please note: The name of a string column can be used everywhere where a string constant is syntactically correct. An error condition will be raised if a numeric column is used in a textual context. All of the above string operators and functions can be arbitrarily nested to form more complex expressions, e.g. `(upper(STRCOL1)[:3]+STRCOL2)[5:]==upper(STRCOL3)`.

2.6 Null value function

Numerical columns in tables can optionally have associated null values. The usual convention is that data entries that are set to a column's special null value are to be interpreted as 'not applicable/available' by the application. The operator function

```
isnull(COLNAME)
```

selects all rows in which the value of `COLNAME` is equal to the column's null value. If the column has no associated null value defined the function evaluates to `false`. Please note:

1. The `isnull()` function can *only* be used in the above manner, i.e., with the name of a numerical table column as argument. Attempts to use it in any other way will lead to an error condition.
2. Unlike integer-valued columns there is always an implicit null value for floating point data, viz. the special IEEE value NaN (Not-A-Number). It is not possible to define any other real value to serve as 'null' for real-valued columns.

2.7 Table column names

Column identifiers can refer to table columns of scalar integral, floating point, boolean, or string type. Mixed-type arithmetic is supported with the exception that names of boolean and string columns must not be used in arithmetic sub-expressions. Names of boolean columns are valid logical sub-expressions, i.e., they do not need to be used in conjunction with the equality operators `==`/`!=` and constants `true`/`false` to form valid logical sub-expressions, e.g. `BOOLCOL==true` is equivalent to `BOOLCOL`.

Vector columns are not yet supported.

2.8 Subexpressions in table attributes

Table attributes being referred to in the selection expression via their names preceded by a hash mark ('`#`') are actually not restricted to contain numeric constants only. In fact the values interpreted as strings can contain anything (including references to other attributes), for instance, to define convenient abbreviations for frequently used constructs. All table attributes will get replaced by their respective values which must yield a valid boolean selection expression. Otherwise a fatal error condition will be raised. Example: If a table contains the attributes:

```
RAWXSQ  = '(RAWX-1)**2'             / RAWX squared
RAWYSQ  = '(RAWY-1)**2'             / RAWY squared
DIST    = 'sqrt(#RAWXSQ + #RAWYSQ)' / distance from center
DISTSEL = '#DIST <= 128'            / select events in circle r=128
```

the selection expression

```
#DISTSEL
```

can be used as an alternative to the full form

```
sqrt((RAWX-1)**2 + (RAWY-1)**2) <= 128
```

2.9 Bitwise operators

For bit manipulations the following six operators are provided:

description	operand	example
bitwise AND	&	b001110110 & b0111 = b110
bitwise OR	|	b001110110 | b0101 = b1110111
bitwise exclusive OR	∧	b001110110 ∧ b0111 = b1110001
left shift	<<	b111 << 2 = b11100
right shift	>>	b111 >> 2 = b1
one's complement (32 bit)	∼	∼b111000 = b111...000111

2.10 Special region filter functions

The following functions evaluate to true if a given point lies inside or on the border of the specified geometric figure:

- `point(x0,y0,Xcolumn,Ycolumn)`
- `line(x0,y0,x1,y1,Xcolumn,Ycolumn)`
- `circle(xCenter,yCenter,radius,Xcolumn,Ycolumn)`
- `sector(xCenter,yCenter,fromAngle,toAngle,Xcolumn,Ycolumn)` or `pie(xCenter,yCenter,fromAngle,toAngle,Xcolumn,Ycolumn)`
- `ring(xCenter,yCenter,radius1,radius2,Xcolumn,Ycolumn)` or `annulus(xCenter,yCenter,radius1,radius2,Xcolumn,Ycolumn)`
- `ellipse(xCenter,yCenter,xHalfWidth,yHalfWidth,rotation,Xcolumn,Ycolumn)`
- `elliptannulus(xCenter,yCenter,xHalfWidthInner,yHalfWidthInner xHalfWidthOuter,yHalfWidthOuter,rotationInner,rotationOuter,Xcolumn,Ycolumn)` or `elliptring(xCenter,yCenter,xHalfWidthInner,yHalfWidthInner xHalfWidthOuter,yHalfWidthOuter,rotationInner,rotationOuter,Xcolumn,Ycolumn)`
- `box(xCenter,yCenter,xHalfWidth,yHalfWidth,rotation,Xcolumn,Ycolumn)`
- `rectangle(xLoLeft,yLoLeft,xUpRight,yUpRight,rotation,Xcolumn,Ycolumn)`
- `rhombus(xCenter,yCenter,xHalfWidth,yHalfWidth,rotation,Xcolumn,Ycolumn)` or `diamond(xCenter,yCenter,xHalfWidth,yHalfWidth,rotation,Xcolumn,Ycolumn)`
- `polygon(x1,y1,x2,y2,x3,y3,x4,y4,...,Xcolumn,Ycolumn)`
- `polygon2(x1,y1,x2,y2,x3,y3,x4,y4,...,Xcolumn,Ycolumn)`

where

`(x0,y0)` : a coordinate pair defining a point - the two points given by `(x0,y0)` and `(Xcolumn,yColumn)` (see below) must coincide for the `point` function to return true

`(x0,y0,x1,y1)` : two points defining a line - the test point `(Xcolumn,Ycolumn)` (see below) must lie on the line to within 0.5 units for the `line` function to return true

`(x1,y1,x2,y2,...)` : the corner points of a polygon

`(xCenter,yCenter)` : the (x,y) position of the center of the region

`(xHalfWidth,yHalfWidth)` : the (x,y) half widths of the region - for `elliptannulus`: values for the inner and outer ellipse respectively

`(xLoLeft,yLoLeft, xUpRight,yUpRight)`	: the coordinates of the lower left and upper right corner of the rectangle
`(radius)`	: the half the diameter of the circle
`(radius1,radius2)`	: the inner and outer half diameters of the annulus
`(fromAngle,toAngle)`	: two angles in decimal degrees defining a sector; `fromAngle` must be lower than or equal to `toAngle`
`(rotation)`	: the angle in decimal degrees that the region is rotated with respect to `(xCenter,yCenter)` except for rectangles which are rotated with respect to the lower left corner (`xLoLeft,yLoLeft`) - elliptical annuli have two angles for the inner and outer ellipse respectively (`rotationInner`, `rotationOuter`)
`(Xcolumn,Ycolumn)`	: the coordinates of the point to test - usually given as column names

Please note:

- Although this is usually the case, the above filter functions do not have to be necessarily applied in the spatial domain. `(Xcolumn,Ycolumn)` can indeed be the names of any columns in the table, i.e., selections in any two-dimensional data space are possible.
- In the case of the `polygon` filter the result of the inclusion test for points that lie exactly on a boundary or coincide with a vertex is uncertain. If this behavior is unacceptable please avail the `polygon2` filter whose inclusion test results for those points is always positive. This comes at the price of a worse run time efficiency.

2.11 Three-dimensional filter functions

The following functions evaluate to true if a vector lies inside or on the border of the specified three dimensional figure:

- `cone(vcen, alpha, vtest)`

where

`vcen`	: vector defining symmetry axis of figure
`alpha`	: half opening angle of cone in rad
`vtest`	: vector for inclusion test

Please note: For the sake of readability and clarity it is recommended to avail these filters in conjunction with the `In` operator (see Sect. 2.13).

2.12 File-based filters

There are three special file-based filters which perform filtering with Good Time Intervals, spatial image masks, and region tables. The syntax is:

`gti(`*`blockspec`*`,Tcolumn)`

`mask(`*`blockspec`*`,Xoffset,Yoffset,Xcolumn,Ycolumn)`

`region(`*`blockspec`*`,Xcolumn,Ycolumn) or region(`*`blockspec`*`)`

with

`Tcolumn`	: the time value to be checked; usually a column name
`Xoffset,Yoffset`	: a vector by which the image mask is to be shifted
`Xcolumn,Ycolumn`	: coordinates of point to test - usually given as column names

blockspec is a block specifier which must point to a GTI table, a mask image, or a region table respectively. It must be in either of the four forms `setname`, `setname+blockid`, `setname[blockid]`, or `setname:blockid` where `setname` must be the name of an existing data set and `blockid` an identifier of a block in that data set. If the first form, `setname`, is used, the filter data are sought in the *first* block of the named data set. For the latter three forms, the data set name is optional. If omitted, it refers to the data set of the currently processed table. *blockid* can either be an identifier starting with a letter which must then be the name of a block in the specified data set or a simple number which is directly interpreted as a block sequence number (starting from 1).[1]

2.12.1 `gti`-filter

The expression `gti(x.gti,`*`t`*`)` evaluates to true if the time t lies within at least one GTI contained in the table `x.gti`. GTI tables must adhere to the OGIP standard [2].

2.12.2 `mask`-filter

`mask(x.msk,`*`x0,y0,x,y`*`)` is true if the value of the mask image shifted by $(x0, y0)$ in the data set `x.msk` at the location (x, y) is not equal to zero.

If the specified mask image contains the *World-Coordinate-System (WCS)*[1] attributes `CRPIX`*`x`*, `CRVAL`*`x`*, `CDELT`*`x`* with *x* being either 1 or 2, image coordinate values are computed from the respective integer pixel numbers n as

$$r_i(n) = (n - \texttt{CRPIX}_i) \times \texttt{CDELT}_i + \texttt{CRVAL}_i, \qquad i = 1, 2 \tag{1}$$

This effectively defines the pixel size in the mask image as `CDELT1` $\times$ `CDELT2`. If symbolic names have been specified in the `mask` filter command as *x* and *y* and the corresponding columns in the processed table have associated WCS-attributes `TCRPX`*`x`*, `TCRVL`*`x`*, and `TCDLT`*`x`* respectively the actual coordinate values are calculated as above. This effectively defines the table pixel size as `TCDLT`$x \times$ `TCDLT`y. The mask filter expression then evaluates to true for each "table pixel" if it has no overlap with any image mask pixel.

Mask images are ordinary rectangular arrays of any integral type supported by the **dal**.

2.12.3 `region`-filter

`region(reg.fits,`*`x,y`*`)` is true if the point (x, y) is contained in any of the regions specified in the region file `reg.fits`.

If the second form of the region filter `region(reg.fits)` is used the coordinates of the test points come from columns with names taken from the `MFORM1` attribute in the specified block.

[1] Examples for valid block specifiers: `gti.fits`, `gti.fits+3`, `gti.fits[GTI23]`, `[98]`, `+GTI10`, `gti.fits:GTI129`

For more informations on this and a formal definition of the region file format see Ref. [3].

2.13 `In`-operator

The `in`-operator is available in three distinct forms

1. `arith IN intervals`
   This tests whether the value to which the arithmetic expression `arith` evaluates lies within at least one among a list of intervals with numeric boundaries. The interval list is either a single interval or a comma separated list of interval specifications. The following table provides an overview of the available intervals types (`x` stands for the value of `arith` above):

interval specification	alternative expression	meaning
`:` or `(:]` or `[:)` or `(:)`	`true`	$-\infty < x < +\infty$
val or `[`*val*`]`	*val* `==` *x*	$x = val$
val`:` or `[`*val*`:]` or `[`*val*`:)`	*val* `<=` *x*	$val <= x < +\infty$
`(`*val*`:]` or `(`*val*`:)`	*val* `<` *x*	$val < x < +\infty$
`:`*val* or `[:`*val*`]` or `(:`*val*`]`	*val* `>=` *x*	$-\infty < x <= val$
`[:`*val*`)` or `(:`*val*`)`	*val* `>` *x*	$-\infty < x < val$
lo`:`*hi* or `[`*lo*`:`*hi*`]`	*lo* `<=` *x* `&&` *hi* `>=` *x*	$lo <= x <= hi$
`(`*lo*`:`*hi*`]`	*lo* `<` *x* `&&` *hi* `>=` *x*	$lo < x <= hi$
`[`*lo*`:`*hi*`)`	*lo* `<=` *x* `&&` *hi* `>` *x*	$lo <= x < hi$
`(`*lo*`:`*hi*`)`	*lo* `<` *x* `&&` *hi* `>` *x*	$lo < x < hi$

   An example of a valid value-in-interval expression which yields true is:

   ```
   3.1415 in :-10,[1:3),3.1,[3.14:3.19),[4:]
   ```

2. `arith in gti(`*gtitable*`)`
   This is equivalent to the expression

   ```
   gti(gtitable, arith)
   ```

3. `(arith1, arith2) in filter(...)`
   where `filter` is either `region` or `mask` (see Sect. 2.12) or any of the region selection functions (see Sect. 2.10). The form is equivalent to

   ```
   filter(..., arith1, arith2)
   ```

   for instance

   ```
   (RAWX, RAWY) in circle(100, 120, 10) == circle(100, 120, 10, RAWX, RAWY)
   ```

   always evaluates to true.

4. `vector in filter(...)`
   where `filter` is a three dimensional region selection function (see Sect. 2.11) and `vector` is a vector for the inclusion test.

2.14 Selected-function

The `selected` function facilitates step-wise filtering of tables. Used as a sub-expression it evaluates to true for a particular row if that rows has been selected in the selection process preceding the current one. If the table undergoes selection for the very first time, `selected` returns true for all table rows, i.e., has no effect. As an example, the expression:

```
SELECTED && PHA >= 10
```

selects all rows which where selected in the previous selection cycle and with PHA values greater than or equal to 10.

2.15 Example expressions

The following paragraphs contain lists of sample expressions which should demonstrate how the various elements described in the above sections can be combined to achieve a desired result:

2.15.1 Table filtering

These sample boolean expressions demonstrate the filtering of an event list table with columns `RAWX`, `RAWY`, `PHA`, `PATTERNID`, `CCDID`, `PHASE`, `TIME`, and `FLAG`:

- `RAWX >= 100 && #ROW != 10`
  select all events with X coordinates greater than or equal to 100; exclude the 10th event anyway
- `RAWX > RAWY`
  select all events to the right of the line `RAWX==RAWY`
- `CIRCLE(354,383,84,RAWX,RAWY)||ELLIPSE(243,215,27,108,0,RAWX,RAWY)`
  select events which lie in the circle or the ellipse with the specified parameters
- `PATTERNID==4`
  select all events with `PATTERNID` equal to 4
- `(CCDID==1 && TIME in gti(ccd1.gti))||(CCDID==2 && TIME in gti(ccd2.gti))`
  select all events from CCD #1 which pass the GTI filters in `ccd1.gti` plus all events from CCD #2 which pass the GTI filters in `ccd2.gti`.
- `PHASE>=.3 && PHASE<=.9`
  select events from the specified phase window
- `(PHA>=100 && PHA<=200)||(PHA>=500 && PHA <=600)`
  select events with PHA values greater than 99 and less than 201 or greater than 499 and less than 601
- `FLAG & b110 != 0`
  select events which have the second and third bit in an (integral) `FLAG` column set
- `SELECTED && PHA >= 10`
  select events which have been selected in the previous selection run and have PHA values greater than or equal to 10

- `PHA in [0:10],(23:40),99,200: && TIME in gti(gti.fits[GTI0])`
  select events with PHA values in the range 0–10, or 23–40 (excl.), or equal to 99, or greater than or equal to 200 and arrival times lying within Good-Time-Intervals contained in the table named `GTI0` in the data set `gti.fits`
- `skyvector(RA, DEC) in cone(#RA_MED, #DEC_MED, 20*#ARCMIN)`
  select rows for which it is true that the J2000 sky vector constructed from the columns `RA` and `DEC` lies within a cone with an opening angle of 20' and a symmetry axis given through the sky vector defined by the two attributes `#RA_MED` and `#DEC_MED`

2.15.2 Table column construction

In the following the symbols `RAWX`, `RAWY` stand for names of existing columns in the table to be processed:

- `sqrt(RAWX**2 + RAWY**2)`
  Constructs a new table column with radius values (provided `RAWX` /`RAWY` contain Cartesian detection coordinates)
- `100. * sin((TIME-#TIMEZERO)/#PERIOD * 360)`
  Construct a "predicted intensity" column for a very simple time-variable source
- `RAWX + 100`
  Construct a new column which contains the `RAWX` coordinates shifted by 100 pixels

2.15.3 Array arithmetic

In the following the symbols `A1`, `A2`, and `A3` stand for arrays with the same dimensionality and equal extents in each dimension:

- `1`
  Construct a new array and initialize all values to 1.
- `A1+A2`
  Add the two arrays `A1` and `A2`.
- `sin(A1+A2)**2 - exp(A3)`
  Construct an array according to the specified expression involving the existing arrays `A1`, `A2`, and `A3`.

2.16 Implementation Status

feature	status	comment
all functionality described in user section v2.0	implemented in **selectlib** v>=3.0	

2.17 TODO

feature	comment
vector columns support	desirable — low priority
matrix algebra support	needed for pyramid filter in `attfilter` task

References

[1] E. W. Greisen and M. Calabretta. Representation of Celestial Coordinates in FITS. *Astron. Astrophys.*, 1996. Found at the URL: `ftp://fits.cv.nrao.edu/fits/documents/wcs/wcs.all.ps`.

[2] W. Pence L. Angelini and A.F. Tennant. The Proposed Timing FITS File Format for High Energy Astrophysics Data. Technical Report OGIP/93-003, NASA/GSFC, Nov 1993. Found at the URL: `http://legacy.gsfc.nasa.gov/docs/heasarc/ofwg/docs/summary/ogip_93_003_summary.html`.

[3] J. McDowell and A. Rots. FITS REGION Binary Table Design. Technical Report ASC-FITS-REGION-1.0, Chandra Science Center, March 1998. Found at the URL: `http://hea-www.harvard.edu/~jcm/asc/docs/asc/region.ps`.