ssclib

June 2, 2019

Abstract

Library of Fortran 90 (and Perl) utilities needed for SAS task development.

1 Description

1.1 Introduction

This library consists of a series of Fortran 90 modules useful for the development of SAS tasks. Each module is described in a separate section below. Each module is contained in a separate file and will be compiled separately. The resulting object files will be combined into a single library file.

The modules divide broadly into (i) those which depend on XMM-specific libraries such as **dal**, **cal** and **errorHandling**, and (ii) those which do not. The only foreign module that members of the second group depend on is called **types** and is at present resident in the sas package **utils**. This small module just defines data types such as int8, bool etc. If it is desired to port the non XMM-specific modules to a non-sas environment it is recommended to also copy this **types** module.

At some stage in the future it may be desirable to move these two groups into separate libraries.

1.2 Angle conventions

All angular variables are assumed to be in radians, unless otherwise indicated. Variable names end in the unit (eg arcsec, deg) if it is not radians.

1.3 Dependency relations:

*** Not yet done this figure.

1.4 Module index

- Section 4: `confusion`:
- Section 5: `constants`:
- Section 6: `coordinate`:
- Section 7: `dal_aux`:
- Section 8: `detection_stats`:
- Section 9: `dss_aux`:
- Section 10: `dump`:
- Section 11: `ellipse`:
- Section 12: `epic_aux`:
- Section 13: `exposure`:
- Section 14: `fftw_aux`:
- Section 15: `geometric_types`:
- Section 16: `histogram_utils`:
- Section 17: `intervals_aux`:
- Section 18: `linear`:
- Section 19: `math_utils`:
- Section 20: `minimizations`:
- Section 21: `oal_aux`:
- Section 22: `parse_list_mod`:
- Section 23: `polygon`:
- Section 24: `psf_ims`:
- Section 25: `random_aux`:
- Section 26: `reallocate`:
- Section 27: `rebinners`:
- Section 28: `regridders`:
- Section 29: `save_image`:
- Section 30: `sort_mod`:
- Section 31: `source_cutouts`:
- Section 32: `splines`:
- Section 33: `ssc_misc`:
- Section 34: `test_utils`:
- Section 35: `warp_utils`:
- Section 36: `wcs_aux`:
- Section 37: `SSCLib` (Perl):
- Section 38: `Coords` (Perl):
- Section 39: `CelCoords` (Perl):

2 General-purpose array-processing utilities

Module name: `array_utils`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)
Masaaki Sakano (University of Leicester, mas@star.le.ac.uk)

2.1 aryStatInfoFlagT structure definition

A set of structure definitions is provided. This consists of one of the elements in type(aryStatInfo???T) (see Sec 2.2), and also is used as an argument in some subroutines/functions in this package. See Sec 2.2 and Table 1 for the (supposed) meaning of each element.

```
type, public :: aryStatInfoFlagT
   integer        :: status = 0 ! Normal
   logical        :: isValLowerUsed = .false., isValUpperUsed = .false. &
      , isAreaLowerUsed = .false., isAreaUpperUsed = .false. &
      , isMaskUsed = .false.
end type aryStatInfoFlagT
```

2.2 aryStatInfo???T structure definition

A set of structure definitions is provided. The above '???' can be Double, Single, Int32, Int16 or Int8. This gives a structure for the statistics for a given (2-d?) array. The following is the example of `aryStatInfoDoubleT`.

```
type, public :: aryStatInfoDoubleT
   real(double)   :: totalsum, mean, sigma
   real(double)   :: realsum
   integer(int32) :: totalentry, validentry
   real(double)   :: minval, maxval
   integer(int32), dimension(:), allocatable :: minindices, maxindices
   real(double)   :: vallower, valupper
   integer(int32), dimension(:), allocatable :: minareaindices, maxareaindices
   type(aryStatInfoFlagT) :: flag
end type aryStatInfoDoubleT
```

This structure is meant to offer the statistical information of an array or its subset. That is, the statistical information for the array, where the valid entry can be filtered based on given

1. external mask file, where True entry is valid,
2. lower and/or upper thresholds for value at each cell,
3. lower and/or upper boundaries (area, if 2-dim) of the indices of the array cell.

If more than one filter condition are given, then the logical product of those conditions are considered, namely, only the entries that satisfy all the given filter conditions are used. Note that the boundary is

Table 1: Elements in `aryStatInfoT`

Element	Type[1]	Description
%totalsum	D/S/I32	Sum (for the valid entry)
%realsum	D	Sum (for the valid entry)
%mean	D (S for SingleT)	Mean
%sigma	D (S for SingleT)	Standard deviation[2]
%totalentry	I32	Total entry (size) of the array.
%validentry	I32	The entry used in calculation.
%minval	D/S/I32/I16/I8	Minimum value among the valid entry.
%maxval	D/S/I32/I16/I8	Minimum value among the valid entry.
%minindices	I32 [Array]	Location (indices: x,y,...) of `minval`
%maxindices	I32 [Array]	Location (indices: x,y,...) of `maxval`
%vallower	D (S for SingleT)	Lower threshold to be a valid entry (if specified).
%valupper	D (S for SingleT)	Upper threshold to be a valid entry (if specified).
%minareaindices	I32 [Array]	Minimum indices for the area of valid entry.
%maxareaindices	I32 [Array]	Maximum indices for the area of valid entry.
%flag%status	Integer	0 if normal.
%flag%isValLowerUsed	Logical	True if `vallower` is used.
%flag%isValUpperUsed	Logical	True if `valupper` is used.
%flag%isAreaLowerUsed	Logical	True if `minareaindices` is used.
%flag%isAreaUpperUsed	Logical	True if `maxareaindices` is used.
%flag%isMaskUsed	Logical	True if an external mask is used.

[1]: D(Double), S(Single), I32/I16/I8 (Int32/Int16/Int8)
[2]: $\sqrt{\frac{1}{N}\sum_i (x_i - \bar{x})^2}$.

inclusive for the second and third conditions. For example, if the lower thresholds for value at each cell is given to be 2.5, then the cells of which the value is smaller than 2.5 are regarded as invalid and are not counted as the valid entry.

Table 1 shows the (supposed) meaning of each element as well as gives the difference in types of elements between the structures (such as, `aryStatInfoDoubleT` and `aryStatInfoInt32T`). Technically each user could give a different meaning for them, but it is discouraged for an obvious reason.

Notes: Some of the elements of the structure can be undefined. For example, if `flag%isAreaLowerUsed` is false, the array `minareaindices` is likely to be undefined, even its size (**NOT** allocated). Similarly `flag%isValLowerUsed` is false, `vallower` is likely to be undefined, and so on. If one tries to access those undefined values that may cause a trouble or even Fortran error, leading to abortion.

The difference between the elements of `totalsum` and `realsum` is the type; the latter is always `double`, whereas the former is `double/single/int32`, depending on the type of the input array; n.b., it is `int32` for any of the integer array. The `totalsum` for an integer array may overflow; in that case `totalsum` has a value `INTEGER_NULL`. Another note is that although the they type of `realsum` for a single-precission float array is `double`, obviously it has a practical accuracy of only `single`-precission.

2.3 AryEdgesInfoT structure definition

Fortran arrays are by default have integer indices, starting from 1. The starting index can be specified by users, however once it is passed to a subroutine, the information is in principle lost. And anyway they still have to be integer each spaced by 1, which is a strong constraint.

In practice a pair of indices in an array $(i, j, ...)$ are given a practice meaning. That information should

ideally be held as an entity of the array – objected-oriented languages may provide some convenient way, but Fortran does not.

This Type variable is designed to hold those information, *e.g.*, lower and upper (integer) bounds of the array, and the physical values corresponding to those 'edges' of the array as follows.

```
type, public :: AryEdgesInfoT
  integer :: aryDimension = -1
  integer,      allocatable :: arySize(:), lboundIndex(:)
  real(double), allocatable :: lEdge(:), uEdge(:)
end type AryEdgesInfoT
```

Note the `aryDimension` gives the rank of the array. It is -1 when uninitialised.

If you want to get the upper bound of the array, use the function `getUbound()` (see Section 2.5.7).

To set an AryEdgesInfoT variable, the function `getAryEdgesInfo()` (see Section 2.8.2) offers a convenient way. You can of course set it by yourself, but if you do it, make sure all the component values in the variable are consistent with one another.

2.4 Integer (index) → scalar integers

2.4.1 Return the axes (array) for the input (i,j) for an array

```
interface getAxesFromIndices
  function getAxesFromIndicesDouble(indices, iLbound, iUbound, lEdge, uEdge) result(axes)
    real(double), intent(in) :: indices(:)
    integer,      intent(in) :: iLbound(size(indices)), iUbound(size(indices))
    real(double), intent(in)           :: lEdge(size(indices))
    real(double), intent(in), optional :: uEdge(size(indices))
    real(double) :: axes(size(indices)) ! result
  end function getAxesFromIndicesDouble

  function getAxesFromIndicesSingle(indices, iLbound, iUbound, lEdge, uEdge) result(axes)
    real(single), intent(in) :: indices(:)
    integer,      intent(in) :: iLbound(size(indices)), iUbound(size(indices))
    real(single), intent(in)           :: lEdge(size(indices))
    real(single), intent(in), optional :: uEdge(size(indices))
    real(single) :: axes(size(indices)) ! result
  end function getAxesFromIndicesSingle

  function getAxesFromIndicesEdgesDouble(indices, aryEdgesInfo) result(axes)
    real(double), intent(in) :: indices(:)
    type(AryEdgesInfoT), intent(in) :: aryEdgesInfo
    real(double) :: axes(size(indices)) ! result
  end function getAxesFromIndicesEdgesDouble

  function getAxesFromIndicesEdgesSingle(indices, aryEdgesInfo) result(axes)
    real(single), intent(in) :: indices(:)
    type(AryEdgesInfoT), intent(in) :: aryEdgesInfo
    real(single) :: axes(size(indices)) ! result
  end function getAxesFromIndicesEdgesSingle
```

```
  end interface
```

In the arguments, `indices(:)` are the coordinates in unit of the index of the array of interest. `i(L|U)bound(:)` are the array of (l—u)bound of the array of interest. `(l|u)Edge(:)` are the array of the lower/upper bounds in unit of physically meaningful values of the array of interest; e.g.,

```
lEdge=(0.5,0.5) uEdge=(256.5,256.5)
```

etc.

If uEdge is not given, it is assumed that the width of axes is the same as the size of the array (= `abs(iUbound-iLbound)` for each axis).

The following is a few examples.

Case 1 The axes for the indices (i, j)=(3, 3) in the array (1:5, 1:5) with the edge (0.5:5.5, 0.5:5.5) is (3.0, 3.0).

Case 2 The axes for the indices (i, j)=(3, 3) in the array (1:5, 1:5) with the edge (2.5:7.5, 2.5:7.5) is (5.0, 5.0).

Case 3 The axes for the indices (i, j)=(3, 3) in the array (1:5, 1:5) with the edge (0.0:10, 0.0:10) is (5, 5).

Case 4 The axes for the indices (i, j)=(3, 3) in the array (1:5, 1:5) with the edge (-10:0.0, -10:0.0) is (-5, -5).

Case 5 The axes for the indices (i, j)=(5, 5) in the array (3:7, 3:7) with the edge (0.0:10, 0.0:10) is (5, 5).

2.4.2 Return the indices (i,j) for the input axes (x,y) of an array

The inverse function of `getAxesFromIndices` (See Section 2.4.1).

```
interface getIndicesFromAxes
  function getIndicesFromAxesDouble(axes, iLbound, iUbound, lEdge, uEdge) result(indices)
    real(double), intent(in) :: axes(:)
    integer,      intent(in) :: iLbound(size(axes)), iUbound(size(axes))
    real(double), intent(in)           :: lEdge(size(axes))
    real(double), intent(in), optional :: uEdge(size(axes))
    real(double) :: indices(size(axes)) ! result
  end function getIndicesFromAxesDouble

  function getIndicesFromAxesSingle(axes, iLbound, iUbound, lEdge, uEdge) result(indices)
    real(single), intent(in) :: axes(:)
    integer,      intent(in) :: iLbound(size(axes)), iUbound(size(axes))
    real(single), intent(in)           :: lEdge(size(axes))
    real(single), intent(in), optional :: uEdge(size(axes))
    real(single) :: indices(size(axes)) ! result
  function getIndicesFromAxesSingle

  function getIndicesFromAxesEdgesDouble(axes, aryEdgesInfo) result(indices)
```

```
    real(double), intent(in) :: axes(:)
    type(AryEdgesInfoT), intent(in) :: aryEdgesInfo
    real(double) :: indices(size(axes)) ! result
  end function getIndicesFromAxesEdgesDouble

  function getIndicesFromAxesEdgesSingle(axes, aryEdgesInfo) result(indices)
    real(single), intent(in) :: axes(:)
    type(AryEdgesInfoT), intent(in) :: aryEdgesInfo
    real(single) :: indices(size(axes)) ! result
  end function getIndicesFromAxesEdgesSingle
end interface
```

Indices, though the returned values here may be Real, mean the index for the given array, therefore for an array

```
ary(int(returned_i), int(returned_j))
```

will give something significant in the Fortran code. Axes are arbitrary and give the frame, which may mean something physical.

If uEdge is not given, it is assumed that the width of axes is the same as the size of the array (= `abs(iUbound-iLbound)` for each axis).

The following is a few examples.

Case 1 The indices for the axes (x,y)=(3.0, 3.0) in the array (1:5, 1:5) with the edge (0.5:5.5, 0.5:5.5) is (3, 3).

Case 2 The indices for the axes (x,y)=(5.0, 5.0) in the array (1:5, 1:5) with the edge (2.5:7.5, 2.5:7.5) is (3, 3).

Case 3 The indices for the axes (x,y)=(5.0, 5.0) in the array (1:5, 1:5) with the edge (0.0:10, 0.0:10) is (3,3).

Case 4 The indices for the axes (x,y)=(-5, -5) in the array (1:5, 1:5) with the edge (-10:0.0, -10:0.0) is (3, 3).

Case 5 The indices for the axes (x,y)=(5.0, 5.0) in the array (3:7, 3:7) with the edge (0.0:10, 0.0:10) is (5, 5).

2.4.3 Calculate the indices(i,j) on the new frame converted from the old frame.

Particularly useful in subroutines.

```
interface calcIndicesFromIndices
  subroutine calcIndicesFromIndicesDblDbl(oldIndices, newIndices, oldLbound, newLbound)
    real(double), intent(in)    :: oldIndices(:)
    real(double), intent(out)   :: newIndices(size(oldIndices))
    integer, intent(in)           :: oldLbound(size(oldIndices))
    integer, intent(in), optional :: newLbound(size(oldIndices))
  end subroutine calcIndicesFromIndicesDblDbl
```

```
  subroutine calcIndicesFromIndicesDblI32(oldIndices, newIndices, oldLbound, newLbound)
    real(double),   intent(in)  :: oldIndices(:)
    integer(int32), intent(out) :: newIndices(size(oldIndices))
    integer, intent(in)          :: oldLbound(size(oldIndices))
    integer, intent(in), optional :: newLbound(size(oldIndices))
  end subroutine calcIndicesFromIndicesDblI32

  subroutine calcIndicesFromIndicesDblI16(oldIndices, newIndices, oldLbound, newLbound)
    real(double),   intent(in)  :: oldIndices(:)
    integer(int16), intent(out) :: newIndices(size(oldIndices))
    integer, intent(in)          :: oldLbound(size(oldIndices))
    integer, intent(in), optional :: newLbound(size(oldIndices))
  end subroutine calcIndicesFromIndicesDblI16

  subroutine calcIndicesFromIndicesSglSgl(oldIndices, newIndices, oldLbound, newLbound)
    real(single),   intent(in)  :: oldIndices(:)
    real(single),   intent(out) :: newIndices(size(oldIndices))
    integer, intent(in)          :: oldLbound(size(oldIndices))
    integer, intent(in), optional :: newLbound(size(oldIndices))
  end subroutine calcIndicesFromIndicesSglSgl

  subroutine calcIndicesFromIndicesSglI32(oldIndices, newIndices, oldLbound, newLbound)
    real(single),   intent(in)  :: oldIndices(:)
    integer(int32), intent(out) :: newIndices(size(oldIndices))
    integer, intent(in)          :: oldLbound(size(oldIndices))
    integer, intent(in), optional :: newLbound(size(oldIndices))
  end subroutine calcIndicesFromIndicesSglI32

  subroutine calcIndicesFromIndicesSglI16(oldIndices, newIndices, oldLbound, newLbound)
    real(single),   intent(in)  :: oldIndices(:)
    integer(int16), intent(out) :: newIndices(size(oldIndices))
    integer, intent(in)          :: oldLbound(size(oldIndices))
    integer, intent(in), optional :: newLbound(size(oldIndices))
  end subroutine calcIndicesFromIndicesSglI16

  subroutine calcIndicesFromIndicesI32I32(oldIndices, newIndices, oldLbound, newLbound)
    integer(int32), intent(in)  :: oldIndices(:)
    integer(int32), intent(out) :: newIndices(size(oldIndices))
    integer, intent(in)          :: oldLbound(size(oldIndices))
    integer, intent(in), optional :: newLbound(size(oldIndices))
  end subroutine calcIndicesFromIndicesI32I32

  subroutine calcIndicesFromIndicesI16I16(oldIndices, newIndices, oldLbound, newLbound)
    integer(int16), intent(in)  :: oldIndices(:)
    integer(int16), intent(out) :: newIndices(size(oldIndices))
    integer, intent(in)          :: oldLbound(size(oldIndices))
    integer, intent(in), optional :: newLbound(size(oldIndices))
  end subroutine calcIndicesFromIndicesI16I16
end interface
```

This calculates the indices `(i,j)` on the new frame converted from those on the old frame. The most likely case is to get a pair of indices of an array in a subroutine/function, all the lbound-s of which must be 1, from the old pair of indices, where the lbound-s can be arbitrary.

`newLbound(:)` is in default 1 (if not given).

The following is an example for a 1-dimension array.

```
integer :: ary(5:8)
ary(7) = -99 ! i=7 is the 3rd element in the array.
call aSub(iPoints=(/7/), lbounds=lbound(ary))

subroutine aSub(iPoints, lbounds)
  integer, intent(in) :: iPoints(:), lbounds(:)
  integer :: outAry(size(iPoints))

  call getIndicesFromIndices(iPoints, outAry, lbounds)
  print *, outAry ! => 3
  ! nb., the old index=7 corresponds to index=3 in the current context.
end subroutine aSub
```

2.5 Array or vector → scalar integers

2.5.1 Return largest true index of a logical vector

```
integer function largestTrueIndex(maskVector)
  logical(bool), intent(in) :: maskVector(:)
end function
```

Examples:

```
largestTrueIndex(((/0, 1, 1/) > 0)) ! returns 3
largestTrueIndex(((/0, 1, 0/) > 0)) ! returns 2
largestTrueIndex(((/0, 0, 0/) > 0)) ! returns 0, ie 1 below vector limit.
```

0 is also returned if the vector is of zero size.

2.5.2 Return smallest true index of a logical vector

```
integer function smallestTrueIndex(maskVector)
  logical(bool), intent(in) :: maskVector(:)
end function
```

Examples:

```
smallestTrueIndex(((/0, 1, 1/) > 0)) ! returns 2
smallestTrueIndex(((/1, 1, 1/) > 0)) ! returns 1
smallestTrueIndex(((/0, 0, 0/) > 0)) ! returns 4, ie 1 above vector limit.
```

size(maskVector)+1 is also returned if the vector is of zero size.

2.5.3 Rectangular bounds of the true area of a logical array

```
subroutine maskArrayLimits(mask, minXi, maxXi, minYi, maxYi)
  logical(bool),  intent(in)            :: mask(:,:)
  integer(int16), intent(out)           :: minXi, maxXi, minYi, maxYi
end subroutine
```

!*** erase the status variable from the actual routine!

Example:

```
logical(bool) :: mask(4,5)

mask(1,:) = ((/0, 0, 0, 0, 0/) > 0)
mask(2,:) = ((/0, 1, 1, 0, 0/) > 0)
mask(3,:) = ((/1, 1, 0, 1, 0/) > 0)
mask(4,:) = ((/1, 0, 0, 0, 0/) > 0)

call maskArrayLimits(mask, minXi, maxXi, minYi, maxYi)
! returns minXi=1, maxXi=4, minYi=2, maxYi=4

mask = .false.

call maskArrayLimits(mask, minXi, maxXi, minYi, maxYi)
! returns minXi=6, maxXi=0, minYi=5, maxYi=0 (ie, nonsense)
```

The same nonsense return occurs if mask is of zero size in either direction. It is up to the user to check that the mask is neither empty nor of zero size before calling maskArrayLimits.

2.5.4 Allocate real to a bin

```
function getBinNumber(xBinEdges, x) result(binNum)
  real(single), intent(in) :: xBinEdges(:), x
  integer(int32)           :: binNum
end function
```

In a commonly encountered situation, one has a set of bins defined by a vector of bin edges xBinEdges, plus a real number x, and it is desired to know into which bin x falls. The present function accomplishes this calculation and returns the index of the bin. The bin edges should occur in increasing order, with no two values the same, but they don't have to be equidistant. The returned value obeys in general the rule

$$xBinEdges(i) \leq x < xBinEdges(i+1) \Rightarrow binNum = i$$

However note the following limiting or pathological cases:

- $x < xBinEdges(1) \Rightarrow binNum = 0$, ie 1 under the lower limit, is returned. It is up to the calling routine to catch these instances if it is necessary to prevent this return value from occurring.

- If $x = xBinEdges(size(xBinEdges))$ it is judged to fall within the last bin and $binNum = size(xBinEdges) - 1$ is returned.
- $x > xBinEdges(size(xBinEdges)) \Rightarrow binNum = size(xBinEdges)$, ie 1 over the upper limit, is returned. It is up to the calling routine to catch these instances if it is necessary to prevent this return value from occurring.

2.5.5 Simpler maxloc

The fortran routine maxloc(array) returns the indices at which the maximum value in array occurs. However the return must be a vector quantity of the same size as the array has dimensions. This can be slightly clunky if array is of dimension 1 - ie, a 1-dimensional vector. The function maxLoc1d allows one to obtain the index of the maximum value of the 1-D argument in a scalar return value. This can save some lines of code.

```
interface maxLoc1d
  function maxLoc1dSingle(vector, mask)
    real(single),  intent(in)           :: vector(:)
    logical(bool), intent(in), optional :: mask(:)
    integer                             :: maxLoc1dSingle
  end function

  function maxLoc1dDouble(vector, mask)
    real(double),  intent(in)           :: vector(:)
    logical(bool), intent(in), optional :: mask(:)
    integer                             :: maxLoc1dSingle
  end function
end interface
```

2.5.6 Simpler minloc

The fortran routine minloc(array) returns the indices at which the minimum value in array occurs. However the return must be a vector quantity of the same size as the array has dimensions. This can be slightly clunky if array is of dimension 1 - ie, a 1-dimensional vector. The function minLoc1d allows one to obtain the index of the minimum value of the 1-D argument in a scalar return value. This can save some lines of code.

```
interface minLoc1d
  function minLoc1dSingle(vector, mask)
    real(single),  intent(in)           :: vector(:)
    logical(bool), intent(in), optional :: mask(:)
    integer                             :: minLoc1dSingle
  end function

  function minLoc1dDouble(vector, mask)
    real(double),  intent(in)           :: vector(:)
    logical(bool), intent(in), optional :: mask(:)
    integer                             :: minLoc1dSingle
  end function
end interface
```

2.5.7 Get ubound

This returns `ubound()` for either a given `aryEdgesInfoT` variable (See Section 2.3) or pair of `lbound` and `size`.

```
interface getUbound
  function getUboundFromEdgesInfo(aryEdgesInfo) result(outUbound)
    type(AryEdgesInfoT), intent(in) :: aryEdgesInfo
    integer :: outUbound(aryEdgesInfo\%aryDimension) ! return
  end function getUboundFromEdgesInfo

  function getUboundFromScalars(inLbound, arySize) result(outUbound)
    integer, intent(in) :: inLbound(:), arySize(size(inLbound))
    integer :: outUbound(size(inLbound)) ! return
  end function getUboundFromScalars
end interface
```

2.6 Array or vector → scalar reals

2.6.1 1-D interpolation

Given a set of piecewise-continuous line segments defined by set of x values (these must be monotonically increasing) and a corresponding set of y values, and given also a single $xSample$ value, this subroutine performs a linear interpolation to return the associated $ySample$ value. If $xSample$ is outside the range of x values, or in other pathological cases, 0 is returned.

```
function linearInterpolate(x, y, xSample) result(ySample)
  real(single), intent(in) :: x(:), y(size(x)), xSample
  real(single) :: ySample
end function linearInterpolate
```

2.6.2 Trapezoid-rule, 1-D, numerical integral

Given two vectors x and y containing samples of a function F, this subroutine returns the 'trapezoidal-rule' estimate of the integral of F. In other words, F is replaced by the set of piecewise-continuous line segments defined by x and y. Note that x is assumed to be monotonically increasing - if it isn't, you'll get strange results.

```
interface trapezoidSum
  function trapezoidSumSingle(x, y) result(summ)
    real(single), intent(in) :: x(:), y(size(x))
    real(single)             :: summ
  end function trapezoidSumSingle

  function trapezoidSumDouble(x, y) result(summ)
    real(double), intent(in) :: x(:), y(size(x))
    real(double)             :: summ
  end function trapezoidSumDouble
end interface
```

2.6.3 Median

This function calculates the median value of the supplied 1- or 2-d array. The function sorts the array values: if the number of elements is odd, the central element of the sorted list is returned; if even, the element in the lower of the two central elements is returned.

```
interface median
  real(single) function medianVector(array)
    real(single), intent(in) :: vector(:)
  end function medianVector

  real(single) function medianArray(array)
    real(single), intent(in) :: array(:,:)
  end function medianArray
end interface
```

2.6.4 Value at histogram fraction

For *binFraction* = 0.5 this gives the same result as median(). The algorithm is as follows: the values in *vector* or *array* are sorted; that element index is identified which, expressed as a fraction of the total number of elements, comes nearest to *binFraction*; finally, the element which occurs at that index of the sorted list is returned.

If *binFraction* is outside the range 0 to 1, the smallest or largest element, whichever is appropriate, is returned.

```
interface valueAtHistoFraction
  real(single) function valueAtHistoFractionVector(vector, binFraction&
    , maskVector)

    real(single),  intent(in)           :: vector(:), binFraction
    logical(bool), intent(in), optional :: maskVector(:)
  end function valueAtHistoFractionVector

  real(single) function valueAtHistoFractionArray(array, binFraction&
    , maskArray)

    real(single),  intent(in)           :: array(:,:), binFraction
    logical(bool), intent(in), optional :: maskArray(:,:)
  end function valueAtHistoFractionArray
end interface
```

2.6.5 sumInt32Dbl

The Fortran90 builtin function `sum()` returns the value in as the same type as its argument (at least in NAG Fortran specification). For example, when `aryInt8` is an array of Integer(int8), the returned value of `sum(aryInt8)` is also Integer(int8) — which is practically not very useful, as the total sum of `aryInt8` is very likely to exceed the maximum (or minimum if negative) possible number of Integer(int8) (which can be obtained with `huge()` function).

This subroutine is developed to avoid that problem; it still returns the total sum of the array as `sum()` does, but always returns the type Integer(int32) and/or Real(double) if requested, whether the type of the argument is int8/16/32. In addition if an overflow happens during the calculation, that is, if the absolute value of the total sum is larger than `huge(int32_variable)`, the returned Integer(int32) are `INTEGER_NULL`, which is defined in `DAL`. Also if the size of the given array is zero, the returned values are `INTEGER_NULL` and `REAL_NULL`.

`sumInt32Dbl()` can accept up to 4-dimensional arrays at the time of writing.

See also Section 2.6.6 for the handier, function version of this routine `sumInt32()`.

```
interface sumInt32Dbl
  subroutine sumInt32DblInt81d(ary, sumInInt32, sumInDbl)
    integer(int8),  intent(in) :: ary(:)
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt81d
  subroutine sumInt32DblInt161d(ary, sumInInt32, sumInDbl)
    integer(int16), intent(in) :: ary(:)
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt161d
  subroutine sumInt32DblInt321d(ary, sumInInt32, sumInDbl)
    integer(int32), intent(in) :: ary(:)
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt321d

  subroutine sumInt32DblInt82d(ary, sumInInt32, sumInDbl)
    integer(int8),  intent(in) :: ary(:,:)
    integer(int32), intent(out), optional :: sumInInt32
  end subroutine sumInt32DblInt82d
  subroutine sumInt32DblInt162d(ary, sumInInt32, sumInDbl)
    integer(int16), intent(in) :: ary(:,:)
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt162d
  subroutine sumInt32DblInt322d(ary, sumInInt32, sumInDbl)
    integer(int32), intent(in) :: ary(:,:)
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt322d

  subroutine sumInt32DblInt83d(ary, sumInInt32, sumInDbl)
    integer(int8),  intent(in) :: ary(:,:,:)
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt83d
  subroutine sumInt32DblInt163d(ary, sumInInt32, sumInDbl)
    integer(int16), intent(in) :: ary(:,:,:)
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt163d
  subroutine sumInt32DblInt323d(ary, sumInInt32, sumInDbl)
    integer(int32), intent(in) :: ary(:,:,:)
```

```
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt323d

  subroutine sumInt32DblInt84d(ary, sumInInt32, sumInDbl)
    integer(int8),  intent(in) :: ary(:,:,:,:)
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt84d
  subroutine sumInt32DblInt164d(ary, sumInInt32, sumInDbl)
    integer(int16), intent(in) :: ary(:,:,:,:)
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt164d
  subroutine sumInt32DblInt324d(ary, sumInInt32, sumInDbl)
    integer(int32), intent(in) :: ary(:,:,:,:)
    integer(int32), intent(out), optional :: sumInInt32
    real(double),   intent(out), optional :: sumInDbl
  end subroutine sumInt32DblInt324d
end interface
```

2.6.6 sumInt32

See Section 2.6.5 (`sumInt32Dbl`) for detail. This routine is the front-end of that, namely the function version, just like the Fortran90 builtin function `sum()` but returns always Integer(int32).

Note that if the returned value is `INTEGER_NULL`, the caller side then can execute `sum(real(ary, double))` in order to get the total sum, in the double-precission float number this time.

```
interface sumInt32
  integer(int32) function sumInt32Int81d(ary)
    integer(int8), intent(in)  :: ary(:)
  end function sumInt32Int81d
  integer(int32) function sumInt32Int82d(ary)
    integer(int8), intent(in)  :: ary(:,:)
  end function sumInt32Int82d
  integer(int32) function sumInt32Int83d(ary)
    integer(int8), intent(in)  :: ary(:,:,:)
  end function sumInt32Int83d
  integer(int32) function sumInt32Int84d(ary)
    integer(int8), intent(in)  :: ary(:,:,:,:)
  end function sumInt32Int84d

  integer(int32) function sumInt32Int161d(ary)
    integer(int16), intent(in)  :: ary(:)
  end function sumInt32Int161d
  integer(int32) function sumInt32Int162d(ary)
    integer(int16), intent(in)  :: ary(:,:)
  end function sumInt32Int162d
  integer(int32) function sumInt32Int163d(ary)
    integer(int16), intent(in)  :: ary(:,:,:)
  end function sumInt32Int163d
  integer(int32) function sumInt32Int164d(ary)
```

```
    integer(int16), intent(in)  :: ary(:,:,:,:)
  end function sumInt32Int164d

  integer(int32) function sumInt32Int321d(ary)
    integer(int32), intent(in)  :: ary(:)
  end function sumInt32Int321d
  integer(int32) function sumInt32Int322d(ary)
    integer(int32), intent(in)  :: ary(:,:)
  end function sumInt32Int322d
  integer(int32) function sumInt32Int323d(ary)
    integer(int32), intent(in)  :: ary(:,:,:)
  end function sumInt32Int323d
  integer(int32) function sumInt32Int324d(ary)
    integer(int32), intent(in)  :: ary(:,:,:,:)
  end function sumInt32Int324d
end interface
```

2.7 Array or vector → array or vector

2.7.1 vectorCross

Performs a vector cross product.

```
  interface vectorCross
    function vectorCrossSingle(vectorA, vectorB)
      real(single), intent(in) :: vectorA(3), vectorB(3)
      real(single)             :: vectorCrossSingle(3)
    end function vectorCrossSingle

    function vectorCrossDouble(vectorA, vectorB)
      real(double), intent(in) :: vectorA(3), vectorB(3)
      real(double)             :: vectorCrossDouble(3)
    end function vectorCrossDouble
  end interface
```

2.7.2 normalizeVector

Given argument $\bar{v}$ returns $\hat{v}$.

```
  interface normalizeVector
    function normalizeVectorSingle(argument)
      real(single), intent(in) :: argument(:)
      real(single) :: normalizeVectorSingle(size(argument))
    end function normalizeVectorSingle

    function normalizeVectorDouble(argument)
      real(double), intent(in) :: argument(:)
      real(double)             :: normalizeVectorDouble(size(argument))
    end function normalizeVectorDouble
  end interface
```

2.7.3 findEdges

```
subroutine findEdges(mask, figureEdges, groundEdges)
  logical(bool), intent(in)  :: mask(:,:)
  logical(bool), intent(out), optional ::&
    figureEdges(size(mask, 1),size(mask, 2)),&
    groundEdges(size(mask, 1),size(mask, 2))
end subroutine findEdges
```

Given an input logical array *mask*, this subroutine returns (optionally) an array $figureEdges$ which is true at all true-valued pixels in *mask* which have at least 1 false-valued pixel among their 8 nearest neighbours. (For pixels at the edges of the input array, the number of nearest neighbours is of course reduced to 5 or 3 as appropriate.)

I give an example as follows, in which false pixels are represented by 0 and true by 1. For an input mask

```
11110000111
11111000011
11111100011
11111100001
00000000000
11111110000
11101111100
```

the returned $figureEdges$ should be

```
00010000110
00011000010
00001100011
11111100001
00000000000
11111110000
00101011100
```

A second array *groundEdges* is also optionally returned, which is equivalent to $figureEdges$ calculated for .not.*mask*.

2.7.4 invertMask

This subroutine inverts the 2-d or 3-d input mask array, which is either Logical, Logical(bool) or Integer(int8), where .true. and .false. are 1 and 0, respectively.

```
interface invertMask
  subroutine invertMask2dLogical(mask)
    logical, intent(inout) :: mask(:,:)
  end subroutine invertMask2dLogical

  subroutine invertMask2dBool(mask)
    logical(bool), intent(inout) :: mask(:,:)
```

```
  end subroutine invertMask2dBool

  subroutine invertMask2dInt8(mask)
    integer(int8), intent(inout) :: mask(:,:)
  end subroutine invertMask2dInt8

  subroutine invertMask3dLogical(mask)
    logical, intent(inout) :: mask(:,:,:)
  end subroutine invertMask3dLogical

  subroutine invertMask3dBool(mask)
    logical(bool), intent(inout) :: mask(:,:,:)
  end subroutine invertMask3dBool

  subroutine invertMask3dInt8(mask)
    integer(int8), intent(inout) :: mask(:,:,:)
  end subroutine invertMask3dInt8
end interface
```

2.8 Array or vector → other

2.8.1 printAryEdgesInfo

This prints to STDOUT the debug information of a given `AryEdgesInfoT` (see Section 2.3).

```
subroutine printAryEdgesInfo(aryEdgesInfo, varName)
  type(AryEdgesInfoT), intent(in) :: aryEdgesInfo
  character(*), intent(in), optional :: varName
end subroutine printAryEdgesInfo
```

2.8.2 getAryEdgesInfo

This returns a `AryEdgesInfoT` (see Section 2.3).

```
interface getAryEdgesInfo
  function getAryEdgesInfoDbl1d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
    integer, parameter :: rankArin = 1
    real(double), intent(in)  :: ary(:)
    real(double), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
    integer,      intent(in), optional :: lboundIndex(rankArin)
    character(*), intent(in), optional :: style
    type(AryEdgesInfoT) :: aryEdgesInfo ! return
  end function getAryEdgesInfoDbl1d

  function getAryEdgesInfoDbl2d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
    integer, parameter :: rankArin = 2
    real(double), intent(in)  :: ary(:,:)
    real(double), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
    integer,      intent(in), optional :: lboundIndex(rankArin)
    character(*), intent(in), optional :: style
```

```
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoDbl2d

function getAryEdgesInfoDbl3d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 3
  real(double), intent(in)  :: ary(:,:,:)
  real(double), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoDbl3d

function getAryEdgesInfoSgl1d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 1
  real(single), intent(in)  :: ary(:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! Start
end function getAryEdgesInfoSgl1d

function getAryEdgesInfoSgl2d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 2
  real(single), intent(in)  :: ary(:,:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoSgl2d

function getAryEdgesInfoSgl3d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 3
  real(single), intent(in)  :: ary(:,:,:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoSgl3d

function getAryEdgesInfoInt321d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 1
  integer(int32), intent(in)  :: ary(:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoInt321d

function getAryEdgesInfoInt322d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 2
  integer(int32), intent(in)  :: ary(:,:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
```

```
end function getAryEdgesInfoInt322d

function getAryEdgesInfoInt323d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 3
  integer(int32), intent(in)  :: ary(:,:,:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoInt323d

function getAryEdgesInfoInt161d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 1
  integer(int16), intent(in)  :: ary(:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoInt161d

function getAryEdgesInfoInt162d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 2
  integer(int16), intent(in)  :: ary(:,:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoInt162d

function getAryEdgesInfoInt163d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 3
  integer(int16), intent(in)  :: ary(:,:,:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoInt163d

function getAryEdgesInfoInt81d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 1
  integer(int8), intent(in)  :: ary(:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoInt81d

function getAryEdgesInfoInt82d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
  integer, parameter :: rankArin = 2
  integer(int8), intent(in)  :: ary(:,:)
  real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
  integer,      intent(in), optional :: lboundIndex(rankArin)
  character(*), intent(in), optional :: style
  type(AryEdgesInfoT) :: aryEdgesInfo ! return
end function getAryEdgesInfoInt82d
```

```
  function getAryEdgesInfoInt83d(ary, lEdge, uEdge, lboundIndex, style) result(aryEdgesInfo)
    integer, parameter :: rankArin = 3
    integer(int8), intent(in)  :: ary(:,:,:)
    real(single), intent(in), optional :: lEdge(rankArin), uEdge(rankArin)
    integer,      intent(in), optional :: lboundIndex(rankArin)
    character(*), intent(in), optional :: style
    type(AryEdgesInfoT) :: aryEdgesInfo ! return
  end function getAryEdgesInfoInt83d
end interface
```

Among the input arguments, `lboundIndex` is `lbound(ary)` (in the caller); or 1 if unspecified. If `style=='wcs'`, then it is allowed neither `lEdge` nor `uEdge` are given (nb., if you for some reason choose to give one, please give the both); in that case `lEdge` is `lboundIndex-0.5` and `uEdge` is accordingly defined. Otherwise `lEdge` and `uEdge` **MUST** be given.

2.8.3 getAryStatInfo

This function returns the structure `aryStatInfo???T` (See Section 2.2), which contains the statistical information of the array. This function offers an interface for five (numerical) types of input Array and the type of the returned value (`aryStatInfo???T`) varies accordingly.

At the time of writing (ssclib-4.6), it accepts the 1- and 2-dimensional arrays. In the future it is planned to accept 3-dimensional arrays as well.

The following is an example interface for the Double-type one. In other types, only the difference is the type of the input Array, `arin` (and the returned type, accordingly – see Section 2.2 for detail).

```
interface getAryStatInfo
  function getAryStatInfoDouble1d(arin, arMaskIn &
     , minAreaIndices, maxAreaIndices, valLower, valUpper, flagInfo) result(retInfo)

    integer, parameter :: rankArin = 1
    type(aryStatInfoDoubleT) :: retInfo
    real(double), intent(in)  :: arin(:)
    logical, intent(in), optional :: arMaskIn(:)
    integer(int32), intent(in), optional :: minAreaIndices(rankArin), maxAreaIndices(rankArin)
    real(double), intent(in), optional :: valLower, valUpper
    type(aryStatInfoFlagT), intent(in), optional :: flagInfo
  end function getAryStatInfoDouble1d

  function getAryStatInfoDouble2d(arin, arMaskIn &
     , minAreaIndices, maxAreaIndices, valLower, valUpper, flagInfo) result(retInfo)

    integer, parameter :: rankArin = 2
    type(aryStatInfoDoubleT) :: retInfo
    real(double), intent(in)  :: arin(:,:)    ! Input data Array
    logical, intent(in), optional :: arMaskIn(:,:)
    integer(int32), intent(in), optional :: minAreaIndices(rankArin), maxAreaIndices(rankArin)
    real(double), intent(in), optional :: valLower, valUpper
    type(aryStatInfoFlagT), intent(in), optional :: flagInfo
  end function getAryStatInfoDouble2d
end interface
```

Note that the ranks of arin and arMaskIn (if specified) have to be identical.

2.8.4 calcAryStatInfoMask

This subroutine is the core routine for the function `getAryStatInfo` (Section 2.8.3). The difference is that this function returns, as well as (`aryStatInfo???T`), the final mask file, which is used to determine the valid entry to calculate the statistical information of the array. If that is what you want, you can call this subroutine directly.

The following is an example interface for the Double-type one. Again, in other types, only the difference is the type of the input Array, `arin` (and the returned type, accordingly – see Section 2.2 for detail).

```
interface calcAryStatInfoMask
  subroutine calcAryStatInfoMaskDouble1d(arin, retInfo, arMaskOut, arMaskIn &
     , minAreaIndices, maxAreaIndices, valLower, valUpper, flagInfo)

    integer, parameter :: rankArin = 1
    real(double), intent(in)  :: arin(:)      ! Input data Array
    type(aryStatInfoDoubleT), intent(out) :: retInfo
    logical, intent(out) :: arMaskOut(:) ! Must be predefined.
    logical, intent(in), optional :: arMaskIn(:)
    integer(int32), intent(in), optional :: minAreaIndices(rankArin), maxAreaIndices(rankArin)
    real(double), intent(in), optional :: valLower, valUpper
    type(aryStatInfoFlagT), intent(in), optional :: flagInfo
  end subroutine calcAryStatInfoMaskDouble1d

  subroutine calcAryStatInfoMaskDouble2d(arin, retInfo, arMaskOut, arMaskIn &
     , minAreaIndices, maxAreaIndices, valLower, valUpper, flagInfo)

    integer, parameter :: rankArin = 2
    real(double), intent(in)  :: arin(:,:)    ! Input data Array
    type(aryStatInfoDoubleT), intent(out) :: retInfo
    logical, intent(out) :: arMaskOut(:,:) ! Must be predefined.
    logical, intent(in), optional :: arMaskIn(:,:)
    integer(int32), intent(in), optional :: minAreaIndices(rankArin), maxAreaIndices(rankArin)
    real(double), intent(in), optional :: valLower, valUpper
    type(aryStatInfoFlagT), intent(in), optional :: flagInfo
  end subroutine calcAryStatInfoMaskDouble2d
end interface
```

Note that the ranks of arin and arMaskIn (if specified) and arMaskOut have to be identical. And (the rank of) arMaskOut has to be defined in the caller side before the call.

2.8.5 printAryStatInfo

Print the contents of `aryStatInfo???T` structure variable to STDOUT. Note that some of the values are not printed if undefined. This returns nothing.

```
interface printAryStatInfo
  subroutine printAryStatInfoDouble(aryInfo)
```

```
    type(aryStatInfo???T), intent(in) :: aryInfo
  end subroutine printAryStatInfoDouble
end interface
```

2.8.6 getAnnularMaskAry

Return a Logical 2-dimensional mask array for a given size, where the area of the pixels at (rInner <= radius < rOuter) are True.

Note: Make sure to deallocate the returned array after use.

```
interface getAnnularMaskAry
  function getAnnularMaskAry(sizeX, sizeY, centX, centY &
     , rOuter, rInner) result(arMask)
    logical, allocatable :: arMask(:,:)
    integer(int32), intent(in) :: sizeX, sizeY
    real(double),   intent(in) :: centX, centY, rOuter
    real(double),   intent(in), optional :: rInner ! 0 in default.
  end function getAnnularMaskAry
end interface
```

3 Subroutine shortcuts for manipulating information used for setting the CAL state

Module name: `cal_aux`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

3.1 Extract information about the instrument, exposure start time and spacecraft and instrument attitude from a dataset header.

```
interface getCalInfo
  subroutine getCalInfoName(setName, instrumentId, expStartTimeStamp&
    , scAttitude, instrumAttitude)

    character(*),                intent(in)  :: setName
    integer(int32),              intent(out) :: instrumentId
    real(double),                intent(out) :: expStartTimeStamp
    type(SpacecraftAttitudeType), intent(out) :: scAttitude, instrumAttitude
  end subroutine getCalInfoName

  subroutine getCalInfoSet(set, instrumentId, expStartTimeStamp, scAttitude&
    , instrumAttitude)

    type(DataSetT),              intent(in)  :: set
    integer(int32),              intent(out) :: instrumentId
    real(double),                intent(out) :: expStartTimeStamp
    type(SpacecraftAttitudeType), intent(out) :: scAttitude, instrumAttitude
```

```
  end subroutine getCalInfoSet
end interface
```

NOTE that this call also sets the state of the **cal** to the returned instrument.

The *instrumentId* and *expStartTimeStamp* are obtained from respectively from the `INSTRUME` and `DATE-OBS` keywords of the dataset header. The *scAttitude* is read from the `RA_PNT`, `DEC_PNT` and `PA_PNT` keywords (this is actually wrong, since the *instrument* attitude is what should be stored in these, not the spacecraft attitude). The boresight is then obtained for the exposure start time, and *instrumAttitude* calculated by applying the boresight rotation to *scAttitude*.

3.2 Spacecraft → instrument attitude

```
function getInstrumentAttitude(scAttitude, timeStamp) result(instrumAttitude)
  real(double),                 intent(in) :: timeStamp
  type(SpacecraftAttitudeType), intent(in) :: scAttitude
  type(SpacecraftAttitudeType)             :: instrumAttitude
end function getInstrumentAttitude
```

NOTE this function requires the **cal** to have been set to the correct instrument before the call.

3.3 Instrument → spacecraft attitude

```
function getScAttitude(instrumAttitude, timeStamp) result(scAttitude)
  real(double),                 intent(in) :: timeStamp
  type(SpacecraftAttitudeType), intent(in) :: instrumAttitude
  type(SpacecraftAttitudeType)             :: scAttitude
end function getScAttitude
```

NOTE this function requires the **cal** to have been set to the correct instrument before the call.

4 Routines to calculate source confusion

Module name: `confusion`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

```
subroutine findConfusedSets(srcX, srcY, srcRadius, confSetNum)
  real(double),   intent(in)  :: srcX(:), srcY(size(srcX))
  real(double),   intent(in)  :: srcRadius(size(srcX))
  integer(int16), intent(out) :: confSetNum(size(srcX))
end subroutine findConfusedSets

  This subroutine takes as inputs a list of source positions $srcX$ and $srcY$ and nominal radii $srcR

  There are no gaps in the returned sequence of confusion indices. Ie for all valid confusion indices
```

5 A module which contains various mathematical and physical constants

Module name: `constants`

Authors: Richard West (University of Leicester, rgw@star.le.ac.uk), Ian Stewart (University of Leicester, ims@star.le.ac.uk).

5.1 Constants

The `constants` module defines a set of widely used mathematical and physical constants. The constants are defined as "double precision" (`real (kind=double)`).

5.1.1 Pi-related constants

Name	Value	Description
`Pi`	3.1415926535897931	π
`TwoPi`	2.0*`Pi`	2π
`FourPi`	4.0*`Pi`	4π
`PiOverTwo`	0.5*`Pi`	$\pi/2$
`PiOn2`	0.5*`Pi`	$\pi/2$
`OneOverPi`	1.0/`Pi`	$1/\pi$
`OneOverTwoPi`	0.5/`Pi`	$1/2\pi$

5.1.2 Angle conversion factors

Name	Value	Description
`DegToRad`	`TwoPi`/360.0	Degrees to radians
`RadToDeg`	360.0/`TwoPi`	Radians to degrees
`Deg2Rad`	`TwoPi`/360.0	Degrees to radians
`PiOn180`	`TwoPi`/360.0	Degrees to radians
`ArcminToRad`	`DegToRad`/60.0	Arcminutes to radians
`RadToArcmin`	`RadToDeg`*60.0	Radians to arcminutes
`ArcsecToRad`	`DegToRad`/3600.0	Arcseconds to radians
`RadToArcsec`	`RadToDeg`*3600.0	Radians to arcseconds

5.1.3 Solid angle conversion factors

Name	Value	Description
`SqDegToSterad`	$\texttt{DegToRad}^2$	Square degrees to steradian
`SqArcminToSterad`	`SqDegToSterad`/3600.0	Square arcminutes to steradians
`SqArcsecToSterad`	`SqDegToSterad`/1.296×10^7	Square arcseconds to steradian
`SteradToSqDeg`	$\texttt{RadToDeg}^2$	Steradian to square degrees
`SteradToSqArcmin`	`SteradToSqDeg`*3600.0	Steradian to square arcminutes
`SteradToSqArcsec`	`SteradToSqDeg`*1.296×10^7	Steradian to square arcseconds

5.1.4 Square roots

Name	Value	Description
RootTwo	1.4142135623730951	$\sqrt{2}$
RootThree	1.7320508075688772	$\sqrt{3}$
RootTen	3.1622776601683795	$\sqrt{10}$
RootPi	1.7724538509055159	$\sqrt{\pi}$

5.1.5 Natural log-related

Name	Value	Description
NapierE	2.7182818284590455	e
lnTwo	0.6931471805599453	$\ln 2$
lnThree	1.0986122886681098	$\ln 3$
lnFive	1.6094379124341003	$\ln 5$
lnSeven	1.9459101490553132	$\ln 7$
lnTen	2.3025850929940459	$\ln 10$
lnPi	1.1447298858494002	$\ln \pi$

5.1.6 Fundamental physical constants

Name	Value	Description
SpeedOfLight	299792458.0	c (m s^{-1})
PlanckH	$6.62606876 \times 10^{-34}$	h (J s)
NewtonG	6.673×10^{-11}	G (m^3 kg^{-1} s^{-2})
ElectronQ	$1.602176462 \times 10^{-19}$	e (C)
Boltzmann	$1.3806503 \times 10^{-23}$	k (J K^{-1})
StefanBoltzmann	5.670400×10^{-8}	σ (W m^{-2} K^{-4})
Avogadro	$6.02214199 \times 10^{23}$	N_A (mol^{-1})

5.1.7 Energy conversion factors

Name	Value	Description
ErgToeV	1.0d-7 / ElectronQ	erg to eV
ErgTokeV	ErgToeV/1000.0	erg to keV
eVToErg	1.0/ErgToev	eV to ergs
keVToErg	1000.0/ErgToev	keV to ergs

5.1.8 Solar system constants

Name	Value	Description
EarthRadius	6378.14	km

5.2 Unit conversion routines

```
function angstroms2eV(angstroms) result(EeV)
```

```
  real(double), intent(in) :: angstroms
  real(double)             :: EeV
end function angstroms2eV

function eV2angstroms(EeV) result(angstroms)
  real(double), intent(in) :: EeV
  real(double)             :: angstroms
end function eV2angstroms
```

5.3 Black Body routines

These subroutines calculate the power (in watts) radiated per unit surface area ($m\hat{2}$) per unit solid angle (sr) by a black body at temperature='kelvin'. The power can be calculated either per unit frequency, at a given frequency (both in hertz), or per unit wavelength (in metres) at a given supplied wavelength (in angstroms).

The subroutines are designed to be portable and as fast yet as accurate as possible.

5.3.1 Per unit frequency

The precision at low frequencies f is limited by the calculation of exp(x)-1, where x = hfk/T. At low values of x, exp(x) is close to 1 and thus the difference between exp(x) and 1 is a number of low precision. Thus for x values less than 0.1, a series expansion of (exp(x)-1)/x is used instead. The minimum precision occurs at x = 0.1 and is approximately equal to precision(1d0)-1.

```
subroutine getBBfluxPerHertz(hertzValues, kelvin, powerValues)
  real(double), intent(in)  :: hertzValues(:), kelvin
  real(double), intent(out) :: powerValues(size(hertzValues))
end subroutine getBBfluxPerHertz

function bbFluxPerHertz(hertz, kelvin) result(power)
  real(double), intent(in) :: hertz, kelvin
  real(double)             :: power
end function bbFluxPerHertz
```

5.3.2 Per unit wavelength

The precision at long wavelengths L is limited by the calculation of exp(x)-1, where x = hk/TL. At low values of x, exp(x) is close to 1 and thus the difference between exp(x) and 1 is a number of low precision. Thus for x values less than 0.1, a series expansion of (exp(x)-1)/x is used instead. The minimum precision occurs at x = 0.1 and is approximately equal to precision(1d0)-1.

```
subroutine getBBfluxPerMetre(angstromValues, kelvin, powerValues)
  real(double), intent(in)  :: angstromValues(:), kelvin
  real(double), intent(out) :: powerValues(size(angstromValues))
end subroutine getBBfluxPerMetre

function bbFluxPerMetre(angstroms, kelvin) result(power)
  real(double), intent(in) :: angstroms, kelvin
```

```
  real(double)             :: power
end function bbFluxPerMetre
```

6 Routines for performing coordinate transforms

**** include changes in version 3.6.5 etc

Module name: `coordinate`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

This module contains subroutines to perform a variety of coordinate transformations. Where possible the **cal** coordinate-transformation routines are used. The present subroutines have been designed to act as wrappers-of-convenience for one or more of the **cal** routines rather than to supplant them.

6.1 getPsfImagePixelCorners

```
subroutine getPsfImagePixelCorners(psfThetaArcsec, psfPhi, psfPixelSizeMm&
  , psfImage, wcs, wcsType, instrumentId, timeStamp, scAttitude&
  , psfImagePixelCorners, psfCentrePixels)

  real(double),                 intent(in)  :: psfThetaArcsec, psfPhi
  type(PsfBinSizeT),            intent(in)  :: psfPixelSizeMm
  real(single),                 intent(in)  :: psfImage(:,:)
  type(WcsT),                   intent(in)  :: wcs
  character(*),                 intent(in)  :: wcsType
  integer(int32),               intent(in)  :: instrumentId
  real(double),                 intent(in)  :: timeStamp
  type(SpacecraftAttitudeType), intent(in)  :: scAttitude
  type(Point2dT),               intent(out) :: psfImagePixelCorners(&
                                                 size(psfImage,1)+1&
                                                ,size(psfImage,2)+1)
  type(Point2dT),               intent(out) :: psfCentrePixels
end subroutine getPsfImagePixelCorners
```

This subroutine takes an image of the Point Spread Function (PSF) returned by the **cal** call CAL_getPsfImage for a given instrument and returns an array of x and y coordinates, in the sky image coordinate system defined by the wcs structure *wcs*, of the intersections of the pixel edge grid of this image. This grid intersections array is necessary as input to the routine regrid (see section **??**), the purpose of which is to rebin the PSF image to sky coordinates.

NOTE this subroutine requires the **cal** to have been set to the correct instrument before the call.

6.2 raDecToInst

```
interface raDecToInst
  subroutine raDecToInstScalar(raDeg, decDeg, scAttitude&
    , timeStamp, detX, detY, thetaArcsec, phi)
```

```
    real(double),                 intent(in) :: raDeg,&
                                                decDeg
    type(SpacecraftAttitudeType), intent(in) :: scAttitude
    real(double),                 intent(in) :: timeStamp
    real(single)     , optional, intent(out) :: detX,&
                                                detY
    real(double)     , optional, intent(out) :: thetaArcsec,&
                                                phi
  end subroutine raDecToInstScalar

  subroutine raDecToInstVector(raDeg, decDeg, scAttitude&
    , timeStamp, detX, detY, thetaArcsec, phi)

    real(double),                 intent(in) :: raDeg(:),&
                                                decDeg(size(raDeg))
    type(SpacecraftAttitudeType), intent(in) :: scAttitude
    real(double),                 intent(in) :: timeStamp
    real(single)     , optional, intent(out) :: detX(size(raDeg)),&
                                                detY(size(raDeg))
    real(double)     , optional, intent(out) :: thetaArcsec(size(raDeg)),&
                                                phi(size(raDeg))
  end subroutine raDecToInstVector

end interface
```

This subroutine converts from celestial coordinates to instrument-centric coordinates, in either the DETX/Y (the same, up to a scalar multiple, as the CAMCOORD2 system) or TELCOORD systems, depending on which of the optional variables *detX*, *detY*, *thetaArcsec* and *phi* the caller has supplied.

NOTE this subroutine requires the **cal** to have been set to the correct instrument before the call.

6.3 instToRaDec

*** There is no subroutine with this name???

```
interface instToRaDec
  subroutine instToRaDecScalar(raDeg, decDeg, scAttitude&
    , timeStamp, detX, detY, thetaArcsec, phi)

    real(double),                 intent(out) :: raDeg,&
                                                 decDeg
    type(SpacecraftAttitudeType), intent(in)  :: scAttitude
    real(double),                 intent(in)  :: timeStamp
    real(single),       optional, intent(in)  :: detX,&
                                                 detY
    real(double),       optional, intent(in)  :: thetaArcsec,&
                                                 phi
  end subroutine instToRaDecScalar

  subroutine instToRaDecVector(raDeg, decDeg, scAttitude&
    , timeStamp, detX, detY, thetaArcsec, phi)
```

```
      real(double),                 intent(out) :: raDeg(:),&
                                                   decDeg(size(raDeg))
      type(SpacecraftAttitudeType), intent(in)  :: scAttitude
      real(double),                 intent(in)  :: timeStamp
      real(single),       optional, intent(in)  :: detX(size(raDeg)),&
                                                   detY(size(raDeg))
      real(double),       optional, intent(in)  :: thetaArcsec(size(raDeg)),&
                                                   phi(size(raDeg))
    end subroutine instToRaDecVector

! instToRaDecArray not yet done

  end interface
```

This subroutine converts to celestial coordinates from instrument-centric coordinates, in either the DETX/Y (the same, up to a scalar multiple, as the CAMCOORD2 system) or TELCOORD systems, depending on which of the optional variables *detX*, *detY*, *thetaArcsec* and *phi* the caller has supplied.

NOTE this function requires the **cal** to have been set to the correct instrument before the call.

6.4 raDecToPixels

```
  interface raDecToPixels
    subroutine raDecToPixelsScalar(ra, dec, wcs, xPixel, yPixel)

      real(double), intent(in)  :: ra,&
                                   dec
      type(WcsT),   intent(in)  :: wcs ! defined in ssclib/wcs_aux
      real(single), intent(out) :: xPixel,&
                                   yPixel
    end subroutine raDecToPixelsScalar

    subroutine raDecToPixelsVector(ra, dec, wcs, xPixel, yPixel)

      real(double), intent(in)  :: ra(:),&
                                   dec(size(ra))
      type(WcsT),   intent(in)  :: wcs ! defined in ssclib/wcs_aux
      real(single), intent(out) :: xPixel(size(ra)),&
                                   yPixel(size(ra))
    end subroutine raDecToPixelsVector

    subroutine raDecToPixelsArray(ra, dec, wcs, xPixel, yPixel)

      real(double), intent(in)  :: ra(:,:),&
                                   dec(size(ra,1),size(ra,2))
      type(WcsT),   intent(in)  :: wcs ! defined in ssclib/wcs_aux
      real(single), intent(out) :: xPixel(size(ra,1),size(ra,2)),&
                                   yPixel(size(ra,1),size(ra,2))
    end subroutine raDecToPixelsArray

  end interface
```

This subroutins returns the pixel coordinates for the given set of the celestial coordinates in degree.

6.5 pixelsToRaDec

```
interface pixelsToRaDec
  subroutine pixelsToRaDecScalar(xPixel, yPixel, wcs, ra, dec)
    real(single), intent(in)  :: xPixel,&
                                 yPixel
    type(WcsT),   intent(in)  :: wcs ! defined in ssclib/wcs_aux
    real(double), intent(out) :: ra,&
                                 dec
  end subroutine pixelsToRaDecScalar

  subroutine pixelsToRaDecVector(xPixel, yPixel, wcs, ra, dec)
    real(single), intent(in)  :: xPixel(:),&
                                 yPixel(size(xPixel))
    type(WcsT),   intent(in)  :: wcs ! defined in ssclib/wcs_aux
    real(double), intent(out) :: ra(size(xPixel)),&
                                 dec(size(xPixel))
  end subroutine pixelsToRaDecVector

  subroutine pixelsToRaDecArray(xPixel, yPixel, wcs, ra, dec)
    real(single), intent(in)  :: xPixel(:,:),&
                                 yPixel(size(xPixel,1),size(xPixel,2))
    type(WcsT),   intent(in)  :: wcs ! defined in ssclib/wcs_aux
    real(double), intent(out) :: ra(size(xPixel,1),size(xPixel,2)),&
                                 dec(size(xPixel,1),size(xPixel,2))
  end subroutine pixelsToRaDecArray(xPixel, yPixel, wcs, ra, dec)
end interface
```

The inverse subroutine of `raDecToPixels()`. This retuns the celestial coordinates (in J2000) in degrees for the given pair of the sky pixel coordinates.

6.6 raDecToTan

```
interface raDecToTan
  subroutine raDecToTanScalar(refRaDeg, refDecDeg, raDeg, decDeg, tanX, tanY)
    real(double), intent(in)  :: refRaDeg,&
                                 refDecDeg,&
                                 raDeg,&
                                 decDeg
    real(double), intent(out) :: tanX,&
                                 tanY
  end subroutine raDecToTanScalar

  subroutine raDecToTanVector(refRaDeg, refDecDeg, raDeg, decDeg, tanX, tanY)
    real(double), intent(in)  :: refRaDeg,&
                                 refDecDeg,&
                                 raDeg(:),&
                                 decDeg(size(raDeg))
    real(double), intent(out) :: tanX(size(raDeg)),&
                                 tanY(size(raDeg))
  end subroutine raDecToTanVector

  subroutine raDecToTanArray(ra, dec, refRa, refDec, xTan, yTan)
```

```
    real(double), intent(in)  :: refRa,&
                                 refDec,&
                                 ra(:,:),&
                                 dec(size(ra,1),size(ra,2))
    real(double), intent(out) :: xTan(size(ra,1),size(ra,2)),&
                                 yTan(size(ra,1),size(ra,2))
  end subroutine raDecToTanArray

  subroutine raDecToTanScalarWcs(ra, dec, wcs, xTan, yTan)
    real(double), intent(in)  :: ra, dec
    type(WcsT),   intent(in)  :: wcs
    real(double), intent(out) :: xTan, yTan
  end subroutine raDecToTanScalarWcs

  subroutine raDecToTanVectorWcs(ra, dec, wcs, xTan, yTan)
    real(double), intent(in)  :: ra(:),&
                                 dec(size(ra))
    type(WcsT),   intent(in)  :: wcs
    real(double), intent(out) :: xTan(size(ra)),&
                                 yTan(size(ra))
  end subroutine raDecToTanVectorWcs

  subroutine raDecToTanArrayWcs(ra, dec, wcs, xTan, yTan)
    real(double), intent(in)  :: ra(:,:),&
                                 dec(size(ra,1),size(ra,2))
    type(WcsT),   intent(in)  :: wcs
    real(double), intent(out) :: xTan(size(ra,1),size(ra,2)),&
                                 yTan(size(ra,1),size(ra,2))
  end subroutine raDecToTanArrayWcs
end interface
```

This transform is a projection from celestial coordinates to that tangent plane normal to the direction defined by `refRaDeg` and `refDecDeg`. The signs of the returned values are such that `tanX` increases in the direction of decreasing `ra` and `tanY` increases in the direction of increasing `dec`. If the tangent plane were viewed from the centre of the celestial sphere, with the celestial north pole at the zenith, `tanX` would increase to rightwards and `tanY` upwards.

6.7 tanToRaDec

```
interface tanToRaDec
  subroutine tanToRaDecScalar(refRaDeg, refDecDeg, tanX, tanY, raDeg, decDeg)
    real(double), intent(in)  :: refRaDeg,&
                                 refDecDeg,&
                                 tanX,&
                                 tanY
    real(double), intent(out) :: raDeg,&
                                 decDeg
  end subroutine tanToRaDecScalar

  subroutine tanToRaDecVector(refRaDeg, refDecDeg, tanX, tanY, raDeg, decDeg)
    real(double), intent(in)  :: refRaDeg,&
                                 refDecDeg,&
                                 tanX(:),&
```

```
                                tanY(size(tanX))
    real(double), intent(out) :: raDeg(size(tanX)),&
                                decDeg(size(tanX))
  end interface

! tanToRaDecArray not yet done
```

This transform is a projection to celestial coordinates from that tangent plane normal to the direction defined by refRaDeg and refDecDeg. The signs of the the returned values are such that tanX increases in the direction of decreasing ra and tanY increases in the direction of increasing dec. If the tangent plane were viewed from the centre of the celestial sphere, with the celestial north pole at the zenith, tanX would increase to rightwards and tanY upwards.

6.8 polarsToRaDec

```
  interface polarsToRaDec
    subroutine polarsToRaDecVector(refRaDeg, refDecDeg, theta, phi, raDeg&
      , decDeg)

      real(double), intent(in)  :: refRaDeg,&
                                   refDecDeg,&
                                   theta(:),&
                                   phi(size(theta))
      real(double), intent(out) :: raDeg(size(theta)),&
                                   decDeg(size(theta))

    end subroutine polarsToRaDecVector
  end interface

! tanToRaDecScalar, tanToRaDecArray not yet done
```

The same as tanToRaDec (section **??**), except the coordinates on the tangent plane are now given in polar coordinates *theta* and *phi* instead of cartesian $tanX$ and $tanY$. The relationship between the two sets is as follows:

$$tanX = \tan(theta)\cos(phi) tanY = \tan(theta)\sin(phi)$$

6.9 instToTan

```
  interface instToTan
    subroutine instToTanScalar(tanX, tanY, refRaDeg, refDecDeg, scAttitude&
      , timeStamp, detX, detY, thetaArcsec, phi)

      real(double),                 intent(out) :: tanX,&
                                                   tanY
      real(double),                 intent(in)  :: refRaDeg,&
                                                   refDecDeg
      type(SpacecraftAttitudeType), intent(in)  :: scAttitude
```

```
    real(double),                 intent(in)  :: timeStamp
    real(single),       optional, intent(in)  :: detX,&
                                                 detY
    real(double),       optional, intent(in)  :: thetaArcsec,&
                                                 phi
  end subroutine instToTanScalar

  subroutine instToTanVector(tanX, tanY, refRaDeg, refDecDeg, scAttitude&
    , timeStamp, detX, detY, thetaArcsec, phi)

    real(double),                 intent(out) :: tanX(:),&
                                                 tanY(size(tanX))
    real(double),                 intent(in)  :: refRaDeg,&
                                                 refDecDeg
    type(SpacecraftAttitudeType), intent(in)  :: scAttitude
    real(double),                 intent(in)  :: timeStamp
    real(single),       optional, intent(in)  :: detX(size(tanX)),&
                                                 detY(size(tanX))
    real(double),       optional, intent(in)  :: thetaArcsec(size(tanX)),&
                                                 phi(size(tanX))
  end subroutine instToTanVector

  subroutine instToTanArray(tanX, tanY, refRaDeg, refDecDeg, scAttitude&
    , timeStamp, detX, detY, thetaArcsec, phi)

    real(double),                 intent(out) :: tanX(:,:),&
                                                 tanY(size(tanX,1),&
                                                      size(tanX,2))
    real(double),                 intent(in)  :: refRaDeg,&
                                                 refDecDeg
    type(SpacecraftAttitudeType), intent(in)  :: scAttitude
    real(double),                 intent(in)  :: timeStamp
    real(single),       optional, intent(in)  :: detX(size(tanX,1),&
                                                      size(tanX,2)),&
                                                 detY(size(tanX,1),&
                                                      size(tanX,2))
    real(double),       optional, intent(in)  :: thetaArcsec(size(tanX,1),&
                                                             size(tanX,2)),&
                                                 phi(size(tanX,1),&
                                                     size(tanX,2))
  end subroutine instToTanArray
end interface
```

Effectively this is just instToRaDec (section **??**) followed by raDecToTan (section **??**).

NOTE this function requires the **cal** to have been set to the correct instrument before the call.

6.10 tanToInst

```
interface tanToInst
  subroutine tanToInstScalar(tanX, tanY, refRaDeg, refDecDeg, scAttitude&
    , timeStamp, detX, detY, thetaArcsec, phi)
```

```
    real(double),                 intent(in)  :: refRaDeg,&
                                                 refDecDeg,&
                                                 tanX,&
                                                 tanY
    type(SpacecraftAttitudeType), intent(in)  :: scAttitude
    real(double),                 intent(in)  :: timeStamp
    real(single),      optional,  intent(out) :: detX,&
                                                 detY
    real(double),      optional,  intent(out) :: thetaArcsec,&
                                                 phi
  end subroutine tanToInstScalar

  subroutine tanToInstVector(tanX, tanY, refRaDeg, refDecDeg, scAttitude&
    , timeStamp, detX, detY, thetaArcsec, phi)

    real(double),                 intent(in)  :: refRaDeg,&
                                                 refDecDeg,&
                                                 tanX(:),&
                                                 tanY(size(tanX))
    type(SpacecraftAttitudeType), intent(in)  :: scAttitude
    real(double),                 intent(in)  :: timeStamp
    real(single),      optional,  intent(out) :: detX(size(tanX)),&
                                                 detY(size(tanX))
    real(double),      optional,  intent(out) :: thetaArcsec(size(tanX)),&
                                                 phi(size(tanX))
  end subroutine tanToInstVector

  subroutine tanToInstArray(tanX, tanY, refRaDeg, refDecDeg, scAttitude&
    , timeStamp, detX, detY, thetaArcsec, phi)

    real(double),                 intent(in)  :: refRaDeg,&
                                                 refDecDeg,&
                                                 tanX(:,:),&
                                                 tanY(size(tanX,1),&
                                                      size(tanX,2))
    type(SpacecraftAttitudeType), intent(in)  :: scAttitude
    real(double),                 intent(in)  :: timeStamp
    real(single),      optional,  intent(out) :: detX(size(tanX,1),&
                                                      size(tanX,2)),&
                                                 detY(size(tanX,1),&
                                                      size(tanX,2))
    real(double),      optional,  intent(out) :: thetaArcsec(size(tanX,1),&
                                                             size(tanX,2)),&
                                                 phi(size(tanX,1),&
                                                     size(tanX,2))
  end subroutine tanToInstArray
end interface
```

Effectively this is just tanToRaDec (section **??**) followed by raDecToInst (section **??**).

NOTE this function requires the **cal** to have been set to the correct instrument before the call.

6.11 instToRaw

```
interface instToRaw
  subroutine instToRawScalarInt16(rawX, rawY, detX, detY, thetaArcsec, phi)
    real(single),   intent(in),  optional :: detX,&
                                             detY
    real(double),   intent(in),  optional :: thetaArcsec,&
                                             phi
    integer(int16), intent(out)           :: rawX,&
                                             rawY
  end subroutine instToRawScalarInt16

  subroutine instToRawVectorInt16(rawX, rawY, detX, detY, thetaArcsec, phi)
    real(single),   intent(in),  optional :: detX(size(rawX)),&
                                             detY(size(rawX))
    real(double),   intent(in),  optional :: thetaArcsec(size(rawX)),&
                                             phi(size(rawX))
    integer(int16), intent(out)           :: rawX(:),&
                                             rawY(size(rawX))
  end subroutine instToRawVectorInt16

  subroutine instToRawScalarReal32(rawXreal, rawYreal, detX, detY&
    , thetaArcsec, phi, isOffChip)

    real(single),  intent(in),  optional :: detX,&
                                            detY
    real(double),  intent(in),  optional :: thetaArcsec,&
                                            phi
    real(single),  intent(out)           :: rawXreal,&
                                            rawYreal
    logical(bool), intent(out), optional :: isOffChip
  end subroutine instToRawScalarReal32

  subroutine instToRawVectorReal32(rawXreal, rawYreal, detX, detY&
    , thetaArcsec, phi, isOffChip)

    real(single),  intent(in),  optional :: detX(size(rawXreal)),&
                                            detY(size(rawXreal))
    real(double),  intent(in),  optional :: thetaArcsec(size(rawXreal)),&
                                            phi(size(rawXreal))
    real(single),  intent(out)           :: rawXreal(:),&
                                            rawYreal(size(rawXreal))
    logical(bool), intent(out), optional :: isOffChip(size(rawXreal))
  end subroutine instToRawVectorReal32
end interface
```

These subroutines convert to chip coordinates (ie, the RAWX/Y or PIXCOORD1 system) from instrument-centric coordinates, the latter being either the DETX/Y (the same, up to a scalar multiple, as the CAMCOORD2 system) or the TELCOORD system, depending on which of the optional variables *detX*, *detY*, *thetaArcsec* and *phi* the caller has supplied.

The 'int16' routines employ the **cal** calls CAL_camCoord1ToChipCoord and CAL_chipCoordToPixCoord1. However, these calls have two drawbacks: firstly, they return integer values, and secondly, they are only valid 'on-chip'. However there are occasions when it is desirable to obtain finer precision in the chip

coordinates and also to be able to out of the strict range. For this reason I wrote the 'real32' routines. The latter do not use the **cal** calls mentioned above. Instead they first move forward by calculating the instrument-centric coordinates of the corners of the CCD; this information is then iused to perform a linear back-transformation of the input instrument-centric coordinates. The logical variable $isOffChip$ is also set.

NOTE this subroutine requires the **cal** to have been set to the correct instrumentId, ccdChipId and (if instrumentId is EMOS1 or EMOS2) ccdNodeId before the call.

6.12 rawToInst (rawToDet)

```
interface rawToInst
  subroutine rawToInstScalar(rawX, rawY, detX, detY, thetaArcsec, phi)
    integer(int16), intent(in)  :: rawX,&
                                   rawY
    real(single),   intent(out) :: detX,&
                                   detY
    real(double),   intent(out) :: thetaArcsec,&
                                   phi
  end subroutine rawToInstScalar

  subroutine rawToInstVector(rawX, rawY, detX, detY, thetaArcsec, phi)
    integer(int16), intent(in)  :: rawX(:),&
                                   rawY(size(rawX))
    real(single),   intent(out) :: detX(size(rawX)),&
                                   detY(size(rawX))
    real(double),   intent(out) :: thetaArcsec(size(rawX)),&
                                   phi(size(rawX))
  end subroutine rawToInstVector

  subroutine rawToInstArray(rawX, rawY, detX, detY, thetaArcsec, phi)
    integer(int16), intent(in)  :: rawX(:,:),&
                                   rawY(size(rawX,1),size(rawX,2))
    real(single),   intent(out) :: detX(size(rawX,1),size(rawX,2)),&
                                   detY(size(rawX,1),size(rawX,2))
    real(double),   intent(out) :: thetaArcsec(size(rawX,1)&
                                              ,size(rawX,2)),&
                                   phi(size(rawX,1),size(rawX,2))
  end subroutine rawToInstArray
end interface
```

These subroutines convert from chip coordinates (ie, the RAWX/Y or PIXCOORD1 system) to instrument-centric coordinates, the TELCOORD system ($thetaArcsec$ and phi), as well as the DETX/Y ($detX$ and $detY$) (the same, up to a scalar multiple, as the CAMCOORD2 system) in unit of pixels, i.e., **not** mm as **cal** supplies. The **cal** calls `CAL_rawXY2mm` and `CAL_camCoord2ToTelCoord` are employed.

NOTE this function requires the **cal** to have been set to the correct instrumentId, ccdChipId and (if instrumentId is EMOS1 or EMOS2) ccdNodeId before the call. This routine does not alter the randomization state of CAL. Hence if you want an identical result every time you call this subroutine,

```
call CAL_setState(randomize=.false.)
```

should be set beforehand.

The subroutine `rawToDet()` is identical to this, except it does not return the TELCOORD system variables (*thetaArcsec* and *phi*).

6.13 getThetaPhiMaps

```
subroutine getThetaPhiMaps(wcs, scAttitude, timeStamp, thetaMap, phiMap)
  type(WcsT),                   intent(in)  :: wcs
  type(SpacecraftAttitudeType), intent(in)  :: scAttitude
  real(double),                 intent(in)  :: timeStamp
  real(double),                 intent(out) :: thetaMap(:,:),&
                                               phiMap(size(thetaMap,1)&
                                                    , size(thetaMap,2))
end subroutine getThetaPhiMaps
```

This subroutine returns two arrays in the sky image coordinate system defined by the WCS structure *wcs*: one containing the θ and the other the ϕ value at that pixel; θ and ϕ being the instrument-mirror-centric TELCOORD-system polar coordinates.

6.14 skyToCartesian

```
interface skyToCartesian
  function skyToCartesianSingle(ra, dec) result(vector)
    real(single), intent(in) :: ra, dec
    real(single)             :: vector(3)
  end function skyToCartesian

  function skyToCartesianDouble(ra, dec) result(vector)
    real(double), intent(in) :: ra, dec
    real(double)             :: vector(3)
  end function skyToCartesian
end interface
```

Returns a vector of direction cosines of the celestial coordinates *ra* and *dec*. NOTE *ra* and *dec* must be in radians.

6.15 cartesianToSky

```
subroutine cartesianToSky(vector, ra, dec)
  real(double), intent(in)  :: vector(3)
  real(double), intent(out) :: ra, dec
end subroutine cartesianToSky
```

Takes a vector of direction cosines and returns the corresponding celestial coordinates *ra* and *dec*. NOTE *ra* and *dec* are in radians.

6.16 telCoordToDetXY

```
interface telCoordToDetXY
  subroutine telCoordToDetXYScalar(thetaArcsec, phi, detX, detY)
    real(double), intent(in)  :: phi, thetaArcsec
    real(single), intent(out) :: detX, detY
  end subroutine telCoordToDetXYScalar

  subroutine telCoordToDetXYVector(thetaArcsec, phi, detX, detY)
    real(double), intent(in)  :: phi(:), thetaArcsec(size(phi))
    real(single), intent(out) :: detX(size(phi)), detY(size(phi))
  end subroutine telCoordToDetXYVector
end interface
```

Takes a position in the TELCOORD system and returns it in DETXY (ie, in CAMCOORD2 multiplied by a factor to convert mm at the focal plane (the unit of CAMCOORD2) to units of 0.05 arcsec (the unit of DETXY)).

6.17 detXYToTelCoord

```
interface detXYToTelCoord
  module procedure detXYToTelCoordScalar
  module procedure detXYToTelCoordVector
end interface
```

Takes a position in the DETXY (ie, in CAMCOORD2 multiplied by a factor to convert mm at the focal plane (the unit of CAMCOORD2) to units of 0.05 arcsec (the unit of DETXY)) system and returns it in TELCOORD.

6.18 detXY unit definition

The coordinate module also contains the following line:

```
real(single), public, parameter :: detUnitArcsec = 0.05
```

A better place for this would arguably be in a .par file somewhere.

6.19 angleBetweenCelCoords

```
interface angleBetweenCelCoords
  function angleBetweenCelCoordsSingle(vectorA, vectorB, isRadian) result(angle)
    real(single) :: angle
    real(single), intent(in) :: vectorA(2), vectorB(2)
    logical, intent(in), optional :: isRadian
  end function angleBetweenCelCoordsSingle
```

```
  function angleBetweenCelCoordsDouble(vectorA, vectorB, isRadian) result(angle)
    real(double) :: angle
    real(double), intent(in) :: vectorA(2), vectorB(2)
    logical, intent(in), optional :: isRadian
  end function angleBetweenCelCoordsDouble
end interface angleBetweenCelCoords
```

Returns the angle between the two pair of input celestial coordinates. The default unit is radian (`isRadian` is T), but can be degree (`isRadian` is F).

7 An additional layer over the DAL which implements some short cuts

Module name: `dal_aux`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

This module contains subroutines designed to augment the Data Access Layer routines for accessing data in FITS files (see the **dal** library). The subroutines perform some short cuts I have found useful.

7.1 splitSetTabName

```
subroutine splitSetTabName(setTabName, setName, tabName, noColonFound, useBlock)
  character(*),  intent(in)            :: setTabName
  character(*),  intent(out)           :: setName, tabName
  logical,       intent(out), optional :: noColonFound
  logical,       intent(in),  optional :: useBlock
end subroutine splitSetTabName
```

The parameter type 'table' (see **param**) accepts a string consisting of a dataset name followed by a colon followed by a binary table name. If the user forgets to include the colon + table name, the resulting **dal** error is not very helpful as an indication of what has gone wrong. Personally I find it more useful to have as the default behaviour in this case that the first table in the dataset should be opened, with an accompanying warning. So I have written this routine splitSetTabName() to act as a trap for the situation in which the user leaves off the colon+table name. The idea is that the string read from a 'table'-type parameter is sent first to splitSetTabName(), which searches the string for a colon; if it finds one, then it returns the before- and after-colon strings respectively in setName and tabName; if no colon is detected, splitSetTabName() issues a warning (if noColonFound is not given), returns the entire string in setName, and also attempts to extract the name of the first table in the dataset (unless useBlock is given and is .false.) and returns this in tabName, which can be an empty string. In the latter case, if the file does not exist and if noColonFound is not given, then it raises an error.

An example of how to use splitSetTabName() is as follows:

```
setTabName = stringParameter('mytable') ! this should be of param type 'table'
call splitSetTabName(setTabName, setName, tabName)
set = dataSet(setName, READ)
tab = table(set, tabName)
```

...

7.2 readArrayData

It is often useful to be able to read an array of any data type into a fortran array of a single data type. The following interface covers just about every combination I could think of.

```
interface readArrayData
  subroutine readArrayDataName1dReal32(imageSetName, vector)
    character(*), intent(in) :: imageSetName
    real(single), pointer    :: vector(:)
  end subroutine readArrayDataName1dReal32

  subroutine readArrayDataName2dReal32(imageSetName, image)
    character(*), intent(in) :: imageSetName
    real(single), pointer    :: image(:,:)
  end subroutine readArrayDataName2dReal32

  subroutine readArrayDataName3dReal32(imageSetName, cube)
    character(*), intent(in) :: imageSetName
    real(single), pointer    :: cube(:,:,:)
  end subroutine readArrayDataName3dReal32

  subroutine readArrayDataName1dReal64(imageSetName, vector)
    character(*), intent(in) :: imageSetName
    real(double), pointer    :: vector(:)
  end subroutine readArrayDataName1dReal64

  subroutine readArrayDataName2dReal64(imageSetName, image)
    character(*), intent(in) :: imageSetName
    real(double), pointer    :: image(:,:)
  end subroutine readArrayDataName2dReal64

  subroutine readArrayDataName3dReal64(imageSetName, cube)
    character(*), intent(in) :: imageSetName
    real(double), pointer    :: cube(:,:,:)
  end subroutine readArrayDataName3dReal64

  subroutine readArrayDataName2dBool(imageSetName, image)
    character(*), intent(in) :: imageSetName
    logical(bool), pointer    :: image(:,:)
  end subroutine readArrayDataName2dBool

  subroutine readArrayDataName2dInt16(imageSetName, image)
    character(*),   intent(in) :: imageSetName
    integer(int16), pointer    :: image(:,:)
  end subroutine readArrayDataName2dInt16

  subroutine readArrayDataName2dInt32(imageSetName, image)
    character(*),   intent(in) :: imageSetName
    integer(int32), pointer    :: image(:,:)
  end subroutine readArrayDataName2dInt32
```

```
  subroutine readArrayDataArray1dReal32(inArray, vector)
    type(ArrayT), intent(in) :: inArray
    real(single), pointer    :: vector(:)
  end subroutine readArrayDataArray1dReal32

  subroutine readArrayDataArray2dReal32(inArray, image)
    type(ArrayT), intent(in) :: inArray
    real(single), pointer    :: image(:,:)
  end subroutine readArrayDataArray2dReal32

  subroutine readArrayDataArray3dReal32(inArray, cube)
    type(ArrayT), intent(in) :: inArray
    real(single), pointer    :: cube(:,:,:)
  end subroutine readArrayDataArray3dReal32

  subroutine readArrayDataArray1dReal64(inArray, vector)
    type(ArrayT), intent(in) :: inArray
    real(double), pointer    :: vector(:)
  end subroutine readArrayDataArray1dReal64

  subroutine readArrayDataArray2dReal64(inArray, image)
    type(ArrayT), intent(in) :: inArray
    real(double), dimension(:,:), pointer :: image
  end subroutine readArrayDataArray2dReal64

  subroutine readArrayDataArray3dReal64(inArray, cube)
    type(ArrayT), intent(in) :: inArray
    real(double), dimension(:,:,:), pointer :: cube
  end subroutine readArrayDataArray3dReal64

  subroutine readArrayDataArray2dBool(inArray, image)
    type(ArrayT), intent(in) :: inArray
    logical(bool), dimension(:,:), pointer :: image
  end subroutine readArrayDataArray2dBool

  subroutine readArrayDataArray2dInt16(inArray, image)
    type(ArrayT), intent(in) :: inArray
    integer(int16), dimension(:,:), pointer :: image
  end subroutine readArrayDataArray2dInt16

  subroutine readArrayDataArray2dInt32(inArray, image)
    type(ArrayT), intent(in) :: inArray
    integer(int32), dimension(:,:), pointer :: image
  end subroutine readArrayDataArray2dInt32
end interface
```

Boolean values are converted to real or integer 0s and 1s; real or integer are converted to boolean TRUE if ¿ 0, FALSE otherwise.

Where the dimensions of the dataset array don't match those of the to-be-returned pointer array, it is eventually intended to convert these as follows:

Dims:	Out 1	Out2	Out 3
In 1	simple	-¿image(1,:)	-¿cube(1,1,:)
In 2	take 1st row	simple	-¿ cube(1,:,:)
In 3	take 1st row, 1st plane	take 1st plane	simple
In ¿3	not supported	not supported	not supported

First rows or planes are always aligned with the biggest dimension(s).

However most of the inter-dimensional functionality is not yet in place.

Note that the returned pointer is not associated with any pointer allocated by a **dal** call such as, for example:

```
arrayDataReal32 => real32Array2Data(inArray)
```

Where readArrayData is called with the name of image dataset, the dataset is released within the subroutine, and all such dataset pointers are at that time deallocated; if readArrayData is called instead with the pointer inArray specified, the dataset and its array remain open, all dataset pointers which were allocated within the subroutine also remain allocated, but deallocate in the normal way at the time the calling routine releases the dataset (or its array). In either case, the returned pointer argument 'vector', 'image', or 'cube' REMAINS ALLOCATED and therefore should be expressely deallocated in the calling routine via the fortran 'deallocate' statement.

7.3 addOrOpenColumn

```
function addOrOpenColumn(tab, colName, dataType, units, comment)&
result(col)
  type(TableT),   intent(in)           :: tab
  character(*),   intent(in)           :: colName
  integer(int32), intent(in), optional :: dataType
  character(*),   intent(in), optional :: units, comment
  type(ColumnT)                        :: col
end function addOrOpenColumn
```

This function opens the column if hasColumn(tab, colName) returns TRUE but adds a new column of this name if not. If a new column is creatd, defaults for the optional arguments 'dataType', 'units' and 'comment' are REAL32, '' and '' respectively.

7.4 readColDataToFixed

This is similar in intention to readArrayData (subsection 7.2). The interface below covers most combinations.

```
interface readColDataToFixed
  subroutine readColToFixedNameReal32(tab, colName, colData)
    type(TableT), intent(in)  :: tab
    character(*), intent(in)  :: colName
    real(single), intent(out) :: colData(:)
  end subroutine readColToFixedNameReal32
```

```
subroutine readColToFixedNameReal64(tab, colName, colData)
  type(TableT), intent(in)  :: tab
  character(*), intent(in)  :: colName
  real(double), intent(out) :: colData(:)
end subroutine readColToFixedNameReal64

subroutine readColToFixedNameInt8(tab, colName, colData)
  type(TableT),  intent(in)  :: tab
  character(*),  intent(in)  :: colName
  integer(int8), intent(out) :: colData(:)
end subroutine readColToFixedNameInt8

subroutine readColToFixedNameInt16(tab, colName, colData)
  type(TableT),   intent(in)  :: tab
  character(*),   intent(in)  :: colName
  integer(int16), intent(out) :: colData(:)
end subroutine readColToFixedNameInt16

subroutine readColToFixedNameInt32(tab, colName, colData)
  type(TableT),   intent(in)  :: tab
  character(*),   intent(in)  :: colName
  integer(int32), intent(out) :: colData(:)
end subroutine readColToFixedNameInt32

subroutine readColToFixedNameStr(tab, colName, colData)
  type(TableT),  intent(in)  :: tab
  character(*),  intent(in)  :: colName
  character(*),  intent(out) :: colData(:)
end subroutine readColToFixedNameStr

subroutine readColToFixedNameBool(tab, colName, colData)
  type(TableT),  intent(in)  :: tab
  character(*),  intent(in)  :: colName
  logical(bool), intent(out) :: colData(:)
end subroutine readColToFixedNameBool

subroutine readColToFixedCol2dReal32(col, colData)
  type(ColumnT), intent(in)  :: col
  real(single),  intent(out) :: colData(:,:)
end subroutine readColToFixedCol2dReal32

subroutine readColToFixedCol2dReal64(col, colData)
  type(ColumnT), intent(in)  :: col
  real(double),  intent(out) :: colData(:,:)
end subroutine readColToFixedCol2dReal64

subroutine readColToFixedColReal32(col, colData)
  type(ColumnT), intent(in)  :: col
  real(single),  intent(out) :: colData(:)
end subroutine readColToFixedColReal32

subroutine readColToFixedColReal64(col, colData)
  type(ColumnT), intent(in)  :: col
  real(double),  intent(out) :: colData(:)
```

```
  end subroutine readColToFixedColReal64

  subroutine readColToFixedColInt8(col, colData)
    type(ColumnT), intent(in)  :: col
    integer(int8), intent(out) :: colData(:)
  end subroutine readColToFixedColInt8

  subroutine readColToFixedColInt16(col, colData)
    type(ColumnT),  intent(in)  :: col
    integer(int16), intent(out) :: colData(:)
  end subroutine readColToFixedColInt16

  subroutine readColToFixedColInt32(col, colData)
    type(ColumnT),  intent(in)  :: col
    integer(int32), intent(out) :: colData(:)
  end subroutine readColToFixedColInt32

  subroutine readColToFixedColStr(col, colData)
    type(ColumnT), intent(in)  :: col
    character(*),  intent(out) :: colData(:)
  end subroutine readColToFixedColStr

  subroutine readColToFixedColBool(col, colData)
    type(ColumnT), intent(in)  :: col
    logical(bool), intent(out) :: colData(:)
  end subroutine readColToFixedColBool
end interface
```

The rules for conversion between datatypes are the same as for readArrayData (subsection 7.2). There is at present no conversion between non-string data and a string-valued 'colData' argument.

Note that the argument 'colData' is NOT a pointer and thus should be made the same size as the column to be read in the calling program.

7.5 readColDataToPtr

These are exactly the same as those routines described in subsection 7.4, except that now the argument 'colData' is a pointer array. This allows the calling routine to avoid having to size it before the call to readColDataToPtr, on the other hand 'colData' should now be DEALLOCATED by the calling program after use. As with readArrayData (subsection 7.2), there is no association between 'colData' and the dataset pointers opened within the subroutine, which are either disassociated within the subroutine (if readColDataToPtr was called with the column name) or at the time the calling program releases the dataset (if readColDataToPtr was called with the column pointer).

```
interface readColDataToPtr
  subroutine readColToPtrName2dReal32(tab, colName, colData)
    type(TableT), intent(in) :: tab
    character(*), intent(in) :: colName
    real(single), pointer    :: colData(:,:)
  end subroutine readColToPtrName2dReal32

  subroutine readColToPtrName2dReal64(tab, colName, colData)
```

```
  type(TableT), intent(in) :: tab
  character(*), intent(in) :: colName
  real(double), pointer    :: colData(:,:)
end subroutine readColToPtrName2dReal64

subroutine readColToPtrNameReal32(tab, colName, colData)
  type(TableT), intent(in) :: tab
  character(*), intent(in) :: colName
  real(single), pointer    :: colData(:)
end subroutine readColToPtrNameReal32

subroutine readColToPtrNameReal64(tab, colName, colData)
  type(TableT), intent(in) :: tab
  character(*), intent(in) :: colName
  real(double), pointer    :: colData(:)
end subroutine readColToPtrNameReal64

subroutine readColToPtrNameInt8(tab, colName, colData)
  type(TableT),  intent(in) :: tab
  character(*),  intent(in) :: colName
  integer(int8), pointer    :: colData(:)
end subroutine readColToPtrNameInt8

subroutine readColToPtrNameInt16(tab, colName, colData)
  type(TableT),   intent(in) :: tab
  character(*),   intent(in) :: colName
  integer(int16), pointer    :: colData(:)
end subroutine readColToPtrNameInt16

subroutine readColToPtrNameInt32(tab, colName, colData)
  type(TableT),   intent(in) :: tab
  character(*),   intent(in) :: colName
  integer(int32), pointer    :: colData(:)
end subroutine readColToPtrNameInt32

subroutine readColToPtrColReal32(col, colData)
  type(ColumnT), intent(in) :: col
  real(single),  pointer    :: colData(:)
end subroutine readColToPtrColReal32

subroutine readColToPtrColReal64(col, colData)
  type(ColumnT), intent(in) :: col
  real(double),  pointer    :: colData(:)
end subroutine readColToPtrColReal64

subroutine readColToPtrColInt8(col, colData)
  type(ColumnT), intent(in) :: col
  integer(int8), pointer    :: colData(:)
end subroutine readColToPtrColInt8

subroutine readColToPtrColInt16(col, colData)
  type(ColumnT),  intent(in) :: col
  integer(int16), pointer    :: colData(:)
end subroutine readColToPtrColInt16
```

```
  subroutine readColToPtrColInt32(col, colData)
    type(ColumnT),  intent(in) :: col
    integer(int32), pointer    :: colData(:)
  end subroutine readColToPtrColInt32
end interface
```

The rules for conversion between datatypes are the same as for readArrayData (subsection 7.2). There is at present no conversion between non-string data and a string-valued 'colData' argument.

7.6 minNonNullValue

This and the following subroutine are useful if you want to find min and max values of a column (at present restricted to REAL32 and REAL64 data types) but have reason to fear that nulls may be present. Real-valued nulls can do funny things to fortran minval() and maxval() functions.

```
interface minNonNullValue
  subroutine minNonNullValueSingle(tab, colName, minValue, allRowsNull)
    type(TableT),  intent(in)  :: tab
    character(*),  intent(in)  :: colName
    real(single),  intent(out) :: minValue
    logical(bool), intent(out) :: allRowsNull
  end subroutine minNonNullValueSingle

  subroutine minNonNullValueDouble(tab, colName, minValue, allRowsNull)
    type(TableT),  intent(in)  :: tab
    character(*),  intent(in)  :: colName
    real(double),  intent(out) :: minValue
    logical(bool), intent(out) :: allRowsNull
  end subroutine minNonNullValueDouble
end interface
```

7.7 maxNonNullValue

```
interface maxNonNullValue
  subroutine maxNonNullValueSingle(tab, colName, maxValue, allRowsNull)
    type(TableT),  intent(in)  :: tab
    character(*),  intent(in)  :: colName
    real(single),  intent(out) :: maxValue
    logical(bool), intent(out) :: allRowsNull
  end subroutine maxNonNullValueSingle

  subroutine maxNonNullValueDouble(tab, colName, maxValue, allRowsNull)
    type(TableT),  intent(in)  :: tab
    character(*),  intent(in)  :: colName
    real(double),  intent(out) :: maxValue
    logical(bool), intent(out) :: allRowsNull
  end subroutine maxNonNullValueDouble
end interface
```

7.8 getDataType

This function returns the dataType (see **dal**) of a given FITS-image or table-column. For the input FITS filename, the form of 'ABC.fits:TABNAME' is allowed, where the TABNAME is the name of the FITS extention of interest. The priority order for the table-name (tabName) in given setTabName, extention number (extNum) and extention name (extName) explicitly given is

$$\text{extName} > \text{tabName} > \text{extNum},$$

that is, extName is always considered first if given. Note that if the given extName (or tabName) is an empty string, it is ignored and the next highest priority one is used.

```
interface getDataType
  function getDataTypeFromName(setTabName, extNum, extName, colName) result(iType)
    integer :: iType
    character(*), intent(in) :: setTabName
    integer, intent(in), optional :: extNum
    character(*), intent(in), optional :: extName, colName
  end function getDataTypeFromName

  function getDataTypeFromSet(set, extNum, extName, colName) result(iType)
    integer :: iType
    type(DataSetT), intent(in) :: set
    integer, intent(in), optional :: extNum
    character(*), intent(in), optional :: extName, colName
  end function getDataTypeFromSet

  function getDataTypeFromTab(tab, colName) result(iType)
    integer :: iType
    type(TableT), intent(in) :: tab
    character(*), intent(in) :: colName
  end function getDataTypeFromTab
end interface
```

7.9 getTypeName

This function returns the string expression for the given Integer as the datatype. This is meant to be used (mainly) in debugging. See `/packages/dal/interface/dal.f90` for detail re the definition.

```
interface getTypeName
  subroutine getTypeName(inInt, outStr)
    integer, intent(in) :: inInt
    character(*), intent(out) :: outStr
  end subroutine getTypeName
end interface
```

7.10 getAttributeTypeName

This function returns the string expression for the given Integer as the data-attribute-type. This is meant to be used (mainly) in debugging. See `/packages/dal/interface/dal.f90` for detail re the definition.

```
interface getAttributeTypeName
  subroutine getAttributeTypeName(inInt, outStr)
    integer, intent(in) :: inInt
    character(*), intent(out) :: outStr
  end subroutine getAttributeTypeName
end interface
```

8 Poissonian statistics and source detectability in this regime

Module name: `detection_stats`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

The subroutines in this module deal broadly with source detection in the Poissonian regime.

8.1 Integrated χ^2 probability

This subroutine gives the probability $P_\chi(\chi^2;\nu)$ of exceeding chi^2 for a given number of degrees of freedom ν, *i.e.*, the single-sided integrated probability, where $P_x(x^2;\nu)$ is the chi^2 probability distribution function (*e.g.*, see Appendix C.4 in Bevington & Robinson (1992, "Data reduction and error analysis for the physical sciences", 2nd edition)).

```
subroutine integratedChi2Prob(chi2, degFree, probability, status)
  real(single), intent(in)           :: chi2
  integer,      intent(in)           :: degFree   ! or real(single)
  real(single), intent(out)          :: probability
  integer,      intent(out), optional :: status
end subroutine integratedChi2Prob
```

The optional argument `status` is returned as follows;

Status	Description
$n(n>0)$	(status in `incompleteGammaQ`; see `math_utils`, Section 1
0	Normal end
-1	When the given chi2 ≤ 0
-2	When the given degFree ≤ 0
$n(n<-2)$	((status in `incompleteGammaQ`) -2; see `math_utils`, Sect

8.2 Poisson probability

This returns the probability $P(i)$ of the occurrence of an integer i according to the Poisson distribution

$$P(i) = \frac{a^i exp(-a)}{i!}$$

where a is the average or expectation value of i.

Note that the argument may also be a real number. In this case what is returned is

$$p(r) = \frac{a^r exp(-a)}{\Gamma(r+1)}$$

The value p is not quite a probability density: it would need to be normalized by

$$\int_0^{\inf} dr \, \frac{a^r exp(-a)}{\Gamma(r+1)}$$

in order for this to be true. However it does have the property that, if $r = i$, $p(r) = P(i)$.

```
interface poissonProb
  real function poissonProbSingle(av, realI)
    real(single), intent(in) :: av, realI
  end function poissonProbSingle

  real function poissonProbInt32(av, i)
    real(single), intent(in) :: av
    integer(int32), intent(in) :: i
  end function poissonProbInt32
end interface
```

8.3 Integrated Poisson probability

This function returns the probability that an integer random variable which obeys a Poisson distribution about an average 'av' will EQUAL OR EXCEED 'i'.

```
real(single) function integratedPoissonProb(av, i)
  real(single), intent(in) :: av
  integer, intent(in) :: i
end function integratedPoissonProb
```

8.4 Source detection limits

The logic of source detection goes as follows. Let us assume to begin with that there is no source at a given location, only background. Let us calculate the probability that the observed counts at that location are due solely to background. If this probability is less than a specified value, our initial assumption was incorrect and there is in fact a source at that location.

The routines described in the present section are designed to take as arguments the probability cutoff (actually a cutoff in likelihood is used) and the background, or expectation value for the counts, and use them to calculate the minimum value of source counts which is detectable at those levels.

To elaborate: given a discrete probability distribution $p(j)$ of event counts j, any sample value c is associated with a certain probability $p_{\rm int}$ (and therefore likelihood $L = -ln(p_{\rm int})$) of it *not* being due to

chance. This probability is obtained by summing the probability values $p(j)$ from $j = c$ to $j = \inf$. For p given by the Poisson distribution, this sum is equal to the incomplete gamma function $P(c, a)$, where a is the expectation value of c. In mathematical terms, the Poissonian likelihood is thus

$$L = -ln[P(c, a)]. \tag{1}$$

(Note: This is ONLY true if the uncertainty in the background or expectation value is insignificant.) The two subroutines described in the present section invert equation 1 to return that value of c which is associated with specified L and a.

8.4.1 Single-band detection

```
interface minDetPoissonCounts
  subroutine minDetPoissonCountsScalar(bkgCount, likelihoodCutoff&
  , detectableSrcCount, detectableSrcCountUncert, status)
    real(single), intent(in)            :: bkgCount, likelihoodCutoff
    real(single), intent(out)           :: detectableSrcCount
    real(single), intent(out)           :: detectableSrcCountUncert
    integer,      intent(out), optional :: status
  end subroutine minDetPoissonCountsScalar
```

8.4.2 Parallel detection over N bands (with no assumptions made about source spectrum)

Here the situation is a little more complicated. If nothing can be assumed about the spectra of the sources, the best detection strategy appears to be as follows:

- Detect in each band separately.
- Calculate likelihood values according to equation 1.
- Add the band likelihoods together for each position.

This sum over likelihoods itself follows a Poisson-like distribution. It can thus be shown that the overall likelihood for any given value of this sum being not due to chance, ie, the overall likelihood $L_{\rm total}$ that there is a source at this position, is given by

$$L_{\rm total} = -ln\{1 - P\Big[f(N), \sum_{i=1}^{N} L_i\Big]\}.$$

where $f(x)$ approximates a linear function of x of slope 1. Monte Carlo studies indicate that $f(2) \sim 2$, $f(5) \sim 4$, $f(10) \sim 8$ and so forth; however **eboxdetect** at the present time assumes that $f(N) = N$; hence that (arguably not quite correct) assumption has been built into the present subroutine as well.

```
  subroutine minDetPoissonCountsVector(bkgCounts, likelihoodCutoff&
  , detectableSrcCounts, detectableSrcCountsUncert, srcCountRatios, status)
```

```
    real(single), intent(in)           :: srcCountRatios(:),&
                                          bkgCounts(size(srcCountRatios)),&
                                          likelihoodCutoff
    real(single), intent(out)          :: detectableSrcCounts(&
                                            size(srcCountRatios))
    real(single), intent(out)          :: detectableSrcCountsUncert(&
                                            size(srcCountRatios))
    integer,      intent(out), optional :: status
  end subroutine minDetPoissonCountsVector
end interface minDetPoissonCounts
```

8.5 Integrated Gaussian probability

This function returns the probability that a gaussian-distributed variable y will depart from the mean $y_{\rm mean}$ by greater than $abs(y - y_{\rm mean})$. For values obeying a gaussian distribution of standard deviation σ, the probability P of a given y value (or greater) occurring by chance is

$$P(y) = 1 - erf\left[abs(y - y_{\rm mean})/\sigma\sqrt{2}\right].$$

```
function integratedGaussProb(testY, meanY, sigma) result(probability)
  real(single), intent(in) :: testY, meanY, sigma
  real(single) :: probability
end function integratedGaussProb
```

8.6 'Degrees of freedom' for a sum of likelihoods

```
function calcChi2HalfDegFree(numLikelihoods) result(f)
  integer(int16), intent(in) :: numLikelihoods
  real(single)               :: f
end function calcChi2HalfDegFree
```

A sum over likelihoods seems to have a probability distribution similar to that of a χ^2 distribution with $2f$ degrees of freedom. The factor f is approximately equal to the number of likelihoods in the sum but not quite. However for the time being (until more accurate information is available, that is) it is assumed that the equality holds exactly.

9 DSS utilities

Module name: `dss_aux`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

The routines are designed to act as an additional layer over the **dsslib** library. They implement many useful short cuts.

Full information about the Data Sub Space (DSS) should of course be sought in the **dsslib** task documentation. However a few explanatory words here would not be out of place.

The function of the DSS is to store information about the criteria used to select events. Thus an XMM event list may contain a DSS if it has been filtered in some way; XMM products such as light curves or spectra, which have been created from event lists, nearly always filtered, may also contain them.

The formal structure of a DSS comprises a list of components, each of which may contain a list of filters. Suuposedly the selection should be reconstructed by ANDing all the filters for each component then ORing all the components, but many filter types (eg GTIs) imply a logical OR internal to the filter.

The DSS is implemented in terms of keywords and extensions, but it is not intended that the user should need to know how the DSS is encoded in these things: the **dsslib** library supplies subroutines to permit basic access to the DSS details without this knowledge.

9.1 Routines which involve the whole DSS

9.1.1 hasDss

Tests whether a dataset, array or table has a Data Sub Space (DSS) attached.

```
logical(bool) function hasDssSet(set)
  type(DataSetT), intent(in) :: set
end function hasDssSet

logical(bool) function hasDssArray(inArray)
  type(ArrayT), intent(in) :: inArray
end function hasDssArray

logical(bool) function hasDssTab(tab)
  type(TableT), intent(in) :: tab
end function hasDssTab

logical(bool) function hasDssBlock(inBlock)
  type(BlockT), intent(in) :: inBlock
end function hasDssBlock
```

9.1.2 dumpDss

Dumps contents of a DSS to STDOUT.

```
subroutine dumpDssBlock(inBlock)
  type(BlockT), intent(in) :: inBlock
end subroutine dumpDssBlock

subroutine dumpDssPointer(dssPointer)
  type(DssT), intent(in) :: dssPointer
end subroutine dumpDssPointer

subroutine dumpDssComponent(dssComp)
  type(DScompT), intent(in) :: dssComp
end subroutine dumpDssComponent
```

9.2 Routines which return information about DSS components

Each DSS component contains several filters. Each filter is associated with the name of the event-list column (called in DSS-speak the 'axis' name) which was filtered (for some filter types, eg REGIONFILTERT, two names are supplied), plus a specification for a set of values of that (or those) column(s).

9.2.1 getNum1stDssCompWithValue

Returns the number of the first component which has a filter of the specified column or 'axis' name which passes the specified value. This searches only filters on single columns.

Note that the component number is meaningful for several **dsslib** calls, eg dssComponent(). Component numbers start from zero.

```
subroutine getNum1stDssCompWithValue(dssPointer, axisName, axisValue&
, firstCompNum, filterType)
  type(DssT),   intent(in) :: dssPointer
  character(*), intent(in) :: axisName
  real(single), intent(in) :: axisValue
  integer, intent(out) :: firstCompNum
  integer, intent(in), optional :: filterType
end subroutine getNum1stDssCompWithValue
```

9.2.2 numDssCompsOfAxis

Returns the number of components which have at least 1 filter on the specified axis. (Note that this searches only filters on single columns.)

```
integer function numDssCompsOfAxis(dssPointer, axisName)
  type(DssT),   intent(in) :: dssPointer
  character(*), intent(in) :: axisName
end function numDssCompsOfAxis
```

9.2.3 numDssCompsWithValue

Returns the number of components which have a filter of the specified column or 'axis' name which passes the specified value. This searches only filters on single columns.

```
integer function numDssCompsWithValue(dssPointer, axisName, axisValue)
  type(DssT),   intent(in) :: dssPointer
  character(*), intent(in) :: axisName
  real(single), intent(in) :: axisValue
end function numDssCompsWithValue
```

9.3 Routines which return information about the filters in a specific component

9.3.1 valueIsPassedByDssComp

Returns TRUE if the component has a filter of the specified column or 'axis' name which passes the specified value. This searches only filters on single columns.

```
logical(bool) function valueIsPassedByDssComp(dssComp, axisName, axisValue)
  type(DScompT),intent(in) :: dssComp
  character(*), intent(in) :: axisName
  real(single), intent(in) :: axisValue
end function valueIsPassedByDssComp
```

9.3.2 anyFilterOfThisAxis

Returns TRUE if the component has a filter of the specified column or 'axis' name. This searches only filters on single columns.

```
logical(bool) function anyFilterOfThisAxis(dssComp, axisName)
  type(DScompT),   intent(in) :: dssComp
  character(*), intent(in) :: axisName
end function anyFilterOfThisAxis
```

9.3.3 numFiltersOfAxis

Returns the number of filters on the specified column or 'axis' name. This searches only filters on single columns.

```
integer function numFiltersOfAxis(dssComp, axisName)
  type(DScompT), intent(in) :: dssComp
  character(*),  intent(in) :: axisName
end function numFiltersOfAxis
```

9.3.4 get1stFilterThisAxis

Returns the handle of the first filter on the specified column or 'axis' name. This searches only filters on single columns.

```
subroutine get1stFilterThisAxis(dssComp, axisName, firstFilter, status)
  type(DScompT),   intent(in)  :: dssComp
  character(*),    intent(in)  :: axisName
  type(DSfilterT), intent(out) :: firstFilter
  integer,         intent(out) :: status
end subroutine get1stFilterThisAxis
```

9.3.5 allFiltersPassValue

This returns true EITHER if the specified component contains no filter on the specified axis OR all of the filters of this component and on this axis pass the specified value.

NOTE: In the original version, it calls `dssFilterName(filter)`, which may cause Segmenation Fault at the time of writing (Apr 2011). Therefore it is rewritten with `dssHasFilter()`. The algorithm is not completely identical, however, in the practical cases it should not cause any trouble, especially if the FITS files do not contain multiple DSS in the same dssComp with the same axis name and filterName (which should not be the case in any FITS file). Note that the original and revised algorithms are switched via a parameter `isUsed_dssFilterName`.

The revised algorithm is as follows:

1. Prepare the array `AryIdFilterT=(/ RANGEFILTERT, REGIONFILTERT, ... /)`. Then, the loop over $i = (1..7)$:
2. Check whether a filter with the given `axisName` for `AryIdFilterT(i)` exists.
3. If so, check the consistency with the given `axisValue` in the filter.
4. Those results are stored in `filterOnThisAxisExists(i)` and `valueIsPassed(i)`. For example, if $i == 2$, then that is for `filterType==REGIONFILTERT` (for the given `dssComp` and `axisName`).
5. Finally returns true/false, as described at the top of this comments here.

```
logical(bool) function allFiltersPassValue(dssComp, axisName, axisValue, filterType)
  type(DScompT),   intent(in) :: dssComp
  character(*), intent(in) :: axisName
  real(single), intent(in) :: axisValue
  integer, intent(in), optional :: filterType
end function allFiltersPassValue
```

9.3.6 valueIsPassedByFilter

Returns TRUE if the specified filter passes the value.

```
logical(bool) function valueIsPassedByFilter(value, filter)
  real(single), intent(in) :: value
  type(DSfilterT), intent(in) :: filter
end function valueIsPassedByFilter
```

9.4 Routines which act on DSS RangeT scalars

The DSS structure type RangeT is described in **dsslib**. It specifies a lower and an upper bound to an interval on the real line, and the types of these bounds. Possible types are INCLUSIVE (which means the bound is included in the interval), EXCLUSIVE (which means the bound is not included in the interval) and UNDEFINED in which case there is no bound at this end of the interval.

Note that the integer constants INCLUSIVE, EXCLUSIVE and UNDEFINED are defined in **dsslib**.

9.4.1 copyRange

Copies one range structure to another.

```
function copyRange(inRange) result(outRange)
  type(RangeT), intent(in) :: inRange
  type(RangeT) :: outRange
end function copyRange
```

9.4.2 checkRangeOverlap

This function compares two ranges to see if they overlap, and if not, which is higher than the other. An integer value is returned, which has the posisble values BOVERLAPSA, BISTOOHIGH, BISTOOLOW. These integer constants are defined in the present module.

```
function checkRangeOverlap(rangeA, rangeB) result(status)
  type(RangeT), intent(in) :: rangeA, rangeB
  integer                  :: status
end function checkRangeOverlap
```

9.4.3 andRangePair

This subroutine takes two overlapping ranges and returns a single range which contains the region of overlap.

Note that the function will not work UNLESS THE RANGES OVERLAP as tested by checkRangeOverlap() (see section 9.4.2).

```
function andRangePair(rangeA, rangeB) result(andedRanges)
  type(RangeT), intent(in) :: rangeA, rangeB
  type(RangeT)             :: andedRanges
end function andRangePair
```

9.4.4 orRangePair

This subroutine takes two overlapping ranges and returns a single range which contains the sum of the two ranges.

Note that the function will not work UNLESS THE RANGES OVERLAP as tested by checkRangeOverlap() (see section 9.4.2).

```
function orRangePair(rangeA, rangeB) result(oredRanges)
  type(RangeT), intent(in) :: rangeA, rangeB
  type(RangeT)             :: oredRanges
end function orRangePair
```

9.4.5 valueIsWithinRange

Returns TRUE if the value is within the specified range.

```
logical(bool) function valueIsWithinRangeSingle(value, range)
  real(single), intent(in) :: value
  type(RangeT), intent(in) :: range
end function valueIsWithinRangeSingle

logical(bool) function valueIsWithinRangeInt32(value, range)
  integer(int32), intent(in) :: value
  type(RangeT), intent(in) :: range
end function valueIsWithinRangeInt32
```

9.5 Routines which act on DSS RangeT vectors

See section 9.4 for some additional information about the **dsslib** RangeT structure type.

The routines in the present section deal with vectors of ranges.

9.5.1 rangesAreWellFormed

A RangeT vector (of size N) is defined as well-formed if and only if it obeys the following conditions:

- Only the lowest and highest bounds of the sequence of ranges are permitted to be of type UNDEFINED.
- For each range for which neither the lower or upper bound is of type UNDEFINED (ie, for each internal range in the sequence), the upper bound value must exceed the lower bound value unless both both bound types are INCLUSIVE, in which case the upper bound value may equal the lower bound value. In other words, each range must encompass some non-empty set of real numbers.
- For all i from 1 to N-1, the upper bound of range i must be less than the lower bound of range i+1, unless both bounds are of type EXCLUSIVE, in which case the upper bound of range i may equal the lower bound of range i+1. In other words, the gap between two adjacent ranges must encompass some non-empty set of real numbers.

Note that any ranges pointer can be brought into valid condition by passing it through the subroutine correctRanges() (section 9.5.4).

```
logical(bool) function rangesAreWellFormed(ranges)
  type(RangeT), intent(in) :: ranges(:)
end function rangesAreWellFormed
```

9.5.2 copyRanges

Copies one vector of ranges to another.

NOTE! You should deallocate the pointer outRanges() after use.

```
  subroutine copyRanges(inRanges, outRanges)
    type(RangeT), intent(in) :: inRanges(:)
    type(RangeT), pointer    :: outRanges(:)
  end subroutine copyRanges
```

9.5.3 readRanges

The function of this subroutine is to return a vector containing all the ranges from all filters on the axis 'axisName'.

NOTE! The returned pointer ranges() should be deallocated after use.

```
%  subroutine readRangesDss(dssPointer, axisName, ranges, dssConstraints)
  subroutine readRangesDss(dssPointer, axisName, ranges)
    type(DssT),   intent(in) :: dssPointer
    character(*), intent(in) :: axisName
    type(RangeT), pointer    :: ranges(:)
%    type(dssConstraintType), intent(in), optional  :: dssConstraints(:)
  end subroutine readRangesDss

  subroutine readRangesComp(dssComp, axisName, ranges)
    type(DScompT), intent(in) :: dssComp
    character(*),  intent(in) :: axisName
    type(RangeT),  pointer    :: ranges(:)
  end subroutine readRangesComp
```

9.5.4 correctRanges

This takes a vector of ranges and returns them in a well-formed sequence (see section 9.5.1). Note that the argument is a pointer because the operation may change the number of elements.

```
  subroutine correctRanges(ranges)
    type(RangeT), pointer :: ranges(:)
  end subroutine correctRanges
```

9.5.5 integrateRanges

This is a function to perform numerical integration of y(x) (using the trapezoid rule) over a set of discrete ranges specified via the xRanges argument.

NOTE! (i) The values in the vector x() should be in increasing order. (ii) The ranges may occur in any order, but are otherwise assumed to be well-formed. (iii) If any upper range bound is undefined, the upper x value is used instead for that range bound; likewise for undefined lower bounds.

!*** should change it so that the ranges are required to be well-formed. This would make the routine less general but makes the accepted properties of ranges simpler.

```
  interface integrateRanges
    function integrateRangesScalar(x, y, xRange) result(approxIntegral)
      real(single), intent(in) :: x(:), y(size(x))
      type(RangeT), intent(in) :: xRange
      real(single) :: approxIntegral
    end function integrateRangesScalar

    function integrateRangesVector(x, y, xRanges) result(approxIntegral)
      real(single), intent(in) :: x(:), y(size(x))
      type(RangeT), intent(in) :: xRanges(:)
      real(single) :: approxIntegral
    end function integrateRangesVector
  end interface
```

9.5.6 andRangesPair

This function takes as input two sequences of ranges and returns a sequence which contains all overlaps between the input ranges.

Points to note:

- The input range sequences must be well-formed (see section 9.5.1). The output is well-formed.
- The function returns a pointer argument. Deallocating this correctly is a little tricky, and I may eventually turn this (and all similar functions) into a subroutine. The function should NOT be called iteratively as in the following example:

  ```
  andedRanges => andRangesPair(andedRanges, rangesB)
  ```

  The memory that the pointer andedRanges pointed to before the call now has no pointer to it, since at the moment the call is executed andedRanges is reassigned to the same memory that the function points to, which was newly assigned during the call. The initial memory pointed to by andedRanges therefore cannot now be deallocated. Better would be:

  ```
  tempAndedRanges => andRangesPair(andedRanges, rangesB)
  deallocate(andedRanges)
  andedRanges => tempAndedRanges
  ```

  Then later, when appropriate, deallocate(andedRanges) for the final time.

```
%  function andRangesPair(rangesA, rangesB, doChecks) result(andedRanges)
  function andRangesPair(rangesA, rangesB) result(andedRanges)
    type(RangeT),  intent(in) :: rangesA(:), rangesB(:)
    type(RangeT),  pointer    :: andedRanges(:)
%    logical(bool), intent(in), optional :: doChecks
  end function andRangesPair
```

9.5.7 orRangesPair

This function takes as input two sequences of ranges and returns a sequence which contains the sum of the input ranges. The returned sequence of ranges is well-formed but, in contrast to the function andRange-

sPair(), the inputs are not required to be well-formed. NOTE however that the same considerations re pointer deallocation also apply here.

```
function orRangesPair(rangesA, rangesB) result(oredRanges)
  type(RangeT), intent(in) :: rangesA(:), rangesB(:)
  type(RangeT), pointer    :: oredRanges(:)
end function orRangesPair
```

9.5.8 dumpRanges

Prints the ranges to standard output.

```
subroutine dumpRanges(ranges)
  type(RangeT), intent(in) :: ranges(:)
end subroutine dumpRanges
```

***** andIntervals

9.6 Routines which deal with DSS GTI filters

GTIs are contained in structures of type IntervalT, which is defined in the module caltypes. This is similar to RangeT except that no type is given for the upper and lower bounds. Where I have translated GTIs into ranges (see for example section 9.6.3) I have taken the lower GTI bound to be INCLUSIVE and the upper to be EXCLUSIVE. A well-formed sequence of GTIs (of size N) should therefore obey the following criteria:

- For each GTI in the sequence, the upper bound value must exceed the lower bound value.
- For all i from 1 to N-1, the upper bound of GTI i must be less than the lower bound of GTI i+1.

There are currently no routines to test or correct the format of non-well-formed sequences of GTIs.

See also section 17.

9.6.1 readGtis

This subroutine extracts from the DSS the sequence of GTIs used to filter events on a specified CCD chip. (The routine assumes that all events which occur on a given CCD chip share a common GTI selection.)

The pointer gti() should be deallocated after use.

```
subroutine readGtis(set, inBlock, ccdNum, gti)
  type(DataSetT),  intent(in) :: set
  type(BlockT),    intent(in) :: inBlock
  integer,         intent(in) :: ccdNum
  type(IntervalT), pointer    :: gti(:)
end subroutine readGtis
```

9.6.2 getGtiFromFilter

This subroutine extracts a GTI sequence from a single DSS filter.

The pointer gti() should be deallocated after use.

```
subroutine getGtiFromFilter(set, filter, gti)
  type(DataSetT),  intent(in) :: set
  type(DSfilterT), intent(in) :: filter
  type(IntervalT), pointer    :: gti(:)
end subroutine getGtiFromFilter
```

9.6.3 andGtis

This function takes as input two sequences of GTIs and returns a sequence which contains all overlaps between the input GTIs. The function makes use of the function andRangesPair() (see section 9.5.6) and similar considerations apply.

```
function andGtis(gtiA, gtiB) result(andedGti)
  type(IntervalT), intent(in) :: gtiA(:), gtiB(:)
  type(IntervalT), pointer    :: andedGti(:)
end function andGtis
```

9.7 Routines which deal with DSS bitmask filters

The format of DSS bit masks is a bit more complicated than it used to be. Basically a bit mask is an integer (in SAS useage usually 32 bits in size), each bit of which is intended to be interpreted in a boolean sense. However the **dsslib** call dssFilterMask() now returns two pointers, onBitMasks and offBitMasks, each of which contains a sequence of masks.

The pointers onBitMasks and offBitMasks are supposed to always be the same size, although there is nothing at the API level to force this to be the case. This is a little unfortunate perhaps and to correct this I have defined in the present module a structure BitMaskT which contains an 'on' mask and an 'off' mask.

9.7.1 getBitMasksFromFilter

This subroutine acts as a wrapper around the **dsslib** routine dssFilterMask(). It retrieves the on and off bit masks from the specified DSS filter, checks that they are the same size, and returns them in a pointer bitMaskPtr of type BitMaskT.

```
subroutine getBitMasksFromFilter(filter, bitMaskPtr)
  type(DSfilterT), intent(in) :: filter
  type(BitMaskT),  pointer    :: bitMaskPtr(:)
end subroutine getBitMasksFromFilter
```

9.7.2 allBitsEquiv

This function tests all the values of the specified bit in the on-masks for logical equivalence and all the values in the off-masks for equivalence and returns TRUE if the equivalence holds. Note that bitNum starts at zero.

Examples:

Element number	on-bits	off-bits
1	0010010101101	1010110101100
2	1110101001110	1001010101001
3	0110010100110	1000110001010
4	0110111001110	0010010100101
5	0010000101101	0000111100011

Taking the right-most bit of each mask to be bit 0, bits 2, 4, 9 and 10 of the on-masks are equivalent, whereas bits 4, 7 and 11 of the off-masks are equivalent. Hence allBitsEquiv() would return TRUE for bitNum=4 but not otherwise.

```
logical(bool) function allBitsEquiv(bitMasks, bitNum)
  type(BitMaskT), intent(in) :: bitMasks(:)
  integer(int32), intent(in) :: bitNum
end function allBitsEquiv
```

10 Subroutines for dumping test output to file or STDOUT

Module name: `dump`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

10.1 String content

Many ASCII codes are not associated with a printable character and thus it can be hard to determine all the characters in a string just by printing it to standard output. The present subroutine takes a string, chops it into individual characters, and prints the ASCII code of each character plus its associated printable character, if there is one.

```
subroutine dumpString(str)
  character(*), intent(in) :: str
end subroutine dumpString
```

10.2 Dumping images to FITS array

It is sometimes convenient to dump a 2-D array to a FITS dataset, without worrying about attributes or data type. The various overloadings of dumpImageToFits allow one to write an array of any numeric data type supported by the **dal**.

```
interface dumpImageToFits
  subroutine dumpImageToFitsSingle(image, setName)
    real(single), intent(in) :: image(:,:)
    character(*), intent(in) :: setName
  end subroutine dumpImageToFitsSingle

  subroutine dumpImageToFitsDouble(image, setName)
    real(double), intent(in) :: image(:,:)
    character(*), intent(in) :: setName
  end subroutine dumpImageToFitsDouble

  subroutine dumpImageToFitsInt8(image, setName)
    integer(int8), intent(in) :: image(:,:)
    character(*), intent(in) :: setName
  end subroutine dumpImageToFitsInt8

  subroutine dumpImageToFitsInt16(image, setName)
    integer(int16), intent(in) :: image(:,:)
    character(*), intent(in) :: setName
  end subroutine dumpImageToFitsInt16

  subroutine dumpImageToFitsInt32(image, setName)
    integer(int32), intent(in) :: image(:,:)
    character(*), intent(in) :: setName
  end subroutine dumpImageToFitsInt32
```

The output data type for the last is actually 8-bit integer:

```
  subroutine dumpImageToFitsBool_temp(image, setName)
    logical(bool), intent(in) :: image(:,:)
    character(*), intent(in) :: setName
  end subroutine dumpImageToFitsBool_temp
end interface dumpImageToFits
```

11 Routines which deal with the geometry of ellipses

Module name: `ellipse`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

This module contains several routines for processing ellipses.

11.1 'Rotated' and 'phase' formats

Dealing with ellipses is complicated by the fact that there are at least two convenient ways to parameterise an ellipse, which I will call the 'rotated' and 'phase' forms. In the 'rotated' form the ellipse is specified by two semiaxes R_x and R_y and a an angle of rotation α. In these terms the ellipse is specified most transparently by three equations:

$$\frac{x_0^2}{R_x^2} + \frac{y_0^2}{R_y^2} = 1 \tag{2}$$
$$x = x_0 \cos(\alpha) - y_0 \sin(\alpha)$$
$$y = y_0 \cos(\alpha) + x_0 \sin(\alpha).$$

The ellipse in 'phase' format is specified by two amplitudes A_x and A_y and a phase ϕ by two parametric equations in θ as follows:

$$x = A_x \cos(\theta) \tag{3}$$
$$y = A_y \cos(\theta + \phi). \tag{4}$$

If an ellipse is thought of as a squashed circle, θ is the angle around the original circle.

Rotations of coordinate system are easily accommodated in the 'rotated' format; changes of aspect ratio of the coordinate system are better accommodated in the 'phase' format.

Subroutines are given for translating between the two formats:

```
subroutine ellipsePhaseToAngle(xAmp, yAmp, phase&
  , shortSemiAxis, longSemiAxis, rotatedAngle)

  real(single), intent(in)  :: xAmp, yAmp, phase
  real(single), intent(out) :: longSemiAxis, shortSemiAxis, rotatedAngle
end subroutine ellipsePhaseToAngle

subroutine ellipseAngleToPhase(shortSemiAxis, longSemiAxis, rotatedAngle&
  , xAmp, yAmp, phase)

  real(single), intent(in)  :: longSemiAxis, shortSemiAxis, rotatedAngle
  real(single), intent(out) :: xAmp, yAmp, phase
end subroutine ellipseAngleToPhase
```

11.2 Is a given point inside a given ellipse?

Two routines are given for the two ellipse formats described above:

```
function pointInEllipseWithPhase(x, y, xAmp, yAmp, phase)
  real(single), intent(in) :: x, y, xAmp, yAmp, phase
  integer :: pointInEllipseWithPhase
end function pointInEllipseWithPhase

function pointInEllipseWithAngle(x, y, xSemiAxis, ySemiAxis, rotatedAngle)
  real(single), intent(in) :: x, y, xSemiAxis, ySemiAxis, rotatedAngle
  integer :: pointInEllipseWithAngle
end function pointInEllipseWithAngle
```

These functions return -1 if the point (x, y) lies fully within the ellipse; 0 if it is on the border; and 1 if it is fully outside the ellipse.

11.3 Generating a set of points along an ellipse locus

```
subroutine calcEllipseFromAngle(xSemi, ySemi, angleDeg, xVals, yVals)
  real(single), intent(in)  :: xSemi, ySemi, angleDeg
  real(single), intent(out) :: xVals(:), yVals(size(xVals))
end subroutine calcEllipseFromAngle
```

Equation 2 can be decomposed into the following pair of parametric equations in θ:

$$x = R_x \cos(\theta)$$

$$y = R_y \sin(\theta).$$

θ here plays the same role as in equations 3 and 4. The subroutine returns points evenly distributed in θ.

The matching 'phase'-style subroutine is

```
subroutine calcEllipseFromPhase(xAmp, yAmp, phase, xVals, yVals)
  real(single), intent(in)  :: xAmp, yAmp, phase
  real(single), intent(out) :: xVals(:), yVals(size(xVals))
end subroutine calcEllipseFromPhase
```

Again the point coordinates returned in xVals and yVals are distributed evenly in θ.

11.4 EllipseT structure definition

A structure definition is provided:

```
type, public :: EllipseT
  real(single) ::&
    xAmp,&
    yAmp,&
    phase,& ! radians
    shortSemiAxis,&
    longSemiAxis,&
    rotatedAngle ! radians
  character(10) :: unit ! eg 'pixels', 'detxy', 'tanxy'
  logical(bool) :: isCircle
end type EllipseT
```

An overloaded subroutine is also provided to initialize a variable of this type:

```
interface initializeEllipseT
  subroutine initializeEllipseTScalar(ellipse)
    type(EllipseT), intent(out) :: ellipse
  end subroutine initializeEllipseTScalar
```

```
  subroutine initializeEllipseTVector(ellipse)
    type(EllipseT), intent(out) :: ellipse(:)
  end subroutine initializeEllipseTVector
end interface
```

For scalar or vector ellipse the values set are:

```
ellipse%xAmp          = 0.0
ellipse%yAmp          = 0.0
ellipse%phase         = 0.0
ellipse%shortSemiAxis = 0.0
ellipse%longSemiAxis  = 0.0
ellipse%rotatedAngle  = 0.0
ellipse%unit          = 'none'
ellipse%isCircle      = .false.
```

12 A module containing routines to give supplementary information about EPIC

Module name: `epic_aux`

Author: Masaaki Sakano (University of Leicester, mas@star.le.ac.uk)

Note that the contents of these routines should be ideally defined somewhere in the library of EPIC.

In the EPIC event files, they use Table names for the exposure in the form of

```
'EXPOSU'//ccdNodeNum
```

where `ccdNodeNum` is a serial number for `ccdNum` (CCD chip number) and `nodeNum` (chip node number for MOSs, of which the default is 1 in almost all the actual observations except for a very few test observations). `ccdNodeNum` is defined and hard-coded in the related tasks as

$$\mathrm{ccdNodeNum} = 10 \times (\mathrm{nodeNumLocal} - 1) + \mathrm{ccdNum}. \quad (5)$$

The following subroutines in this module give this relation.

```
function getCcdNodeNum(ccdNum, nodeNum) result(ccdNodeNum)
  integer(int8), intent(in) :: ccdNum
  integer(int8), intent(in), optional :: nodeNum
end function getCcdNodeNum

subroutine inverseCcdNodeNum(ccdNodeNum, ccdNum, nodeNum, instrumentId)
  integer(int32), intent(in) :: ccdNodeNum
  integer(int8),  intent(out) :: ccdNum
  integer(int8),  intent(out), optional :: nodeNum
  integer(int32), intent(in),  optional :: instrumentId
end subroutine inverseCcdNodeNum
```

In the latter `inverseCcdNodeNum()`, if `instrumentId` is not given, `CAL` is read and is essential.

13 A module containing routines to perform calculations relating to exposure issues

Module name: `exposure`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

14 Some utilities and definitions for applications which make use of the FFTW library

Module name: `fftw_aux`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

This module is meant to be used in conjunction with the Fast Fourier Transform package fftw which comes with the sas. The module contains a header in which some useful variables are declared, as well as the following routine:

```
function findNextHighest2357multiple(i) result(result)
  integer, intent(in) :: i
end function findNextHighest2357multiple
```

This function is designed to look for the smallest integer that satisfies the following conditions: (i) it is greater than or equal to the argument i; (ii) it is a product only of the numbers 2, 3, 5 and 7. The fftw transform works most efficiently on arrays which have dimensions which are products of small primes.

15 A module containing some useful type definitions

Module name: `geometric_types`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

The module defines the following types:

```
type, public :: Point2dT
  real(single) :: x, y
end type

type, public :: Point2dDbleT
  real(double) :: x, y
end type
```

16 Routines for constructing histograms

Module name: `histogram_utils`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

This module is still fairly undeveloped so I won't document it yet.

17 Routines to manipulate data of type IntervalT (see caltypes)

Module name: `intervals_aux`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

The routines in this module offer ways to manipulate data structures of type IntervalT, which is defined in **caltypes**. These seem to be primarily of use for handling vectors of GTIs.

17.1 intervalsAreWellFormed

```
logical(bool) function intervalsAreWellFormed(intervals)
  type(IntervalT), intent(in) :: intervals(:)
end function intervalsAreWellFormed
```

Many of the other functions in the present section don't work unless the intervals are 'well-formed'. I define a well-formed vector of type IntervalT as obeying two conditions: (i) for each interval, the lower value must be ¡ the upper; (ii) the upper value of interval i must be ¡ the lower value of interval i+1. The function returns FALSE if either condition is disobeyed.

See section 9.5.1 for analogous conditions on structures of RangeT type.

17.2 isWithinInterval

```
logical(bool) function isWithinInterval(time, intervals, includeBoundary)
  real(double),    intent(in)           :: time
  type(IntervalT), intent(in)           :: intervals(:)
  logical(bool),   intent(in), optional :: includeBoundary
end function isWithinInterval
```

The argument 'time' is tested to see if it falls within any of the intervals. If optional argument 'includeBoundary' is included and set to TRUE, interval boundaries are considered: that is, for example, if time=intervals(i)

NOTE! This function requires the intervals to be well-formed (see section 17.1).

17.3 andIntervals

```
interface andIntervals
  subroutine andIntervalsBothScalar(intervalA, intervalB, andedIntervals)
```

```
    type(IntervalT), intent(in) :: intervalA, intervalB
    type(IntervalT), pointer    :: andedIntervals(:)
  end subroutine andIntervalsBothScalar

  subroutine andIntervalsOneVector(intervalA, intervalsB, andedIntervals)
    type(IntervalT), intent(in) :: intervalA, intervalsB(:)
    type(IntervalT), pointer    :: andedIntervals(:)
  end subroutine andIntervalsOneVector

  subroutine andIntervalsBothVector(intervalsA, intervalsB, andedIntervals)
    type(IntervalT), intent(in) :: intervalsA(:), intervalsB(:)
    type(IntervalT), pointer    :: andedIntervals(:)
  end subroutine andIntervalsBothVector
end interface
```

In all cases the intervals are first converted to RangeT structures as follows:

```
range%lower%type  = INCLUSIVE
range%upper%type  = EXCLUSIVE
range%lower%value = interval%lower
range%upper%value = interval%upper
```

In this form, they can be ANDed together by use of the dss_aux call andRangesPair (see section 9.5.6)

NOTE! This function requires the intervals to be well-formed (see section 17.1).

17.4 orIntervals

```
interface orIntervals
  subroutine orIntervalsBothScalar(intervalA, intervalB, oredIntervals)
    type(IntervalT), intent(in) :: intervalA, intervalB
    type(IntervalT), pointer    :: oredIntervals(:)
  end subroutine orIntervalsBothScalar

  subroutine orIntervalsOneVector(intervalA, intervalsB, oredIntervals)
    type(IntervalT), intent(in) :: intervalA, intervalsB(:)
    type(IntervalT), pointer    :: oredIntervals(:)
  end subroutine orIntervalsOneVector

  subroutine orIntervalsBothVector(intervalsA, intervalsB, oredIntervals)
    type(IntervalT), intent(in) :: intervalsA(:), intervalsB(:)
    type(IntervalT), pointer    :: oredIntervals(:)
  end subroutine orIntervalsBothVector
end interface
```

In all cases the intervals are first converted to RangeT structures as follows:

```
range%lower%type  = INCLUSIVE
```

```
range%upper%type  = EXCLUSIVE
range%lower%value = interval%lower
range%upper%value = interval%upper
```

In this form, they can be ORed together by use of the dss_aux call orRangesPair (see section 9.5.7)

NOTE! This function requires the intervals to be well-formed (see section 17.1).

18 Least-squares fitting routines

Module name: `linear`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

Contains some routines to do with least-squares estimation and solution of linear equations.

18.1 stdDev

```
function stdDev(vector)
  real(single), intent(in) :: vector(:)
  real(single)             :: stdDev
end function stdDev
```

This find the average $< v >$ of the input values v_i, then estimates the scatter or standard deviation σ of these values from

$$\sigma^2 = \frac{1}{N-1}\sum_{i}^{N}(v_i - < v >)^2.$$

18.2 fitLine

```
subroutine fitLine(x, y, intercept, slope, variance, covar, status)
  real(single), intent(in)            :: x(:), y(size(x))
  real(single), intent(out)           :: intercept, slope
  real(single), intent(out), optional :: variance, covar(2, 2)
  integer,      intent(out), optional :: status
end subroutine fitLine
```

This fits a straight line to the set of points defined by x and y. The solution method is the standard one which assumes uncertainty in the y values only and solves the normal equations to arrive at a solution which minimizes the sum of the squares of the y-separation between the resulting line and each point.

18.3 fitPolynomial

```
subroutine fitPolynomial(x, y, coeffs, yVar, pinMask, chi2, errMatrix, status)
  real(single),  intent(in)            :: x(:),&
                                          y(size(x))
  real(single),  intent(inout)         :: coeffs(:)
  real(single),  intent(in),  optional :: yVar(size(x))
  logical(bool), intent(in),  optional :: pinMask(size(coeffs))
  real(single),  intent(out), optional :: chi2,&
                                          errMatrix(size(coeffs),&
                                          size(coeffs))
  integer,       intent(out), optional :: status
end subroutine fitPolynomial
```

This subroutine fits a polynomial to the set of points defined by x and y. The order of the polynomial is given by the size of the vector 'coeffs'. As per usual, only the y values are assumed to have significant uncertainties. The subroutine solves normal equations to arrive at a solution which minimizes χ^2 between the data and the fitted polynomial.

Some or all of the coefficients can be 'pinned' or not fitted. These values should be supplied in the vector 'coeff' (note that this is of intent 'inout' as required). The appropriate members of 'pinMask' should be set to TRUE; all other members of 'pinMask' should of course be FALSE. For example, suppose it was desired to fit to the data a function of the form $y(x) = a + cx^2$. This is equivalent to fitting a full quadratic function to the data, but with the linear coefficient pinned at 0. To achieve this result, 'coeff' and 'pinMask' should be of size 3, with the following values set:

```
coeff(2) = 0.0
pinMask = /(.false. .true. .false.)/
```

The general form of the normal equations in the event of pinning is perhaps best illustrated by using the above example. In this case the equations are

Note that the matrix of uncertainties in the fitted coefficients as well as the χ^2 value at the optimum are also returned.

18.4 solveLinearTriDiag

18.5 solveLinearEquations

18.6 invertPosDefMatrix

19 Miscellaneous mathematical utilities

Module name: `math_utils`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

20 Function minimization routines

Module name: `minimizations`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

21 Helper subroutines for handling ODF (OAL)

Module name: `oal_aux`

Author: Masaaki Sakano (University of Leicester, mas@star.le.ac.uk)

In any of the following task, the environmental variable `SAS_ODF` should be properly set before the call.

21.1 printODFProposal

Dumps the proposal information derived from the ODF.

```
interface printODFProposal
  subroutine printODFProposal(proposalInfo, printHeader)
    type(ProposalInfoType), intent(in) :: proposalInfo
    logical, intent(in), optional      :: printHeader
  end subroutine printODFProposal
end interface
```

The optional parameter `printHeader` specifies whether the header is also printed to STDOUT (T) or not (F). The default is True.

Note that `proposalInfo` is obtained via

```
call OAL_proposalInfo(proposalInfo)
```

22 List Parsing

Module name: `parse_list_mod`

Author: Dean Hinshaw (NASA/GFSC, dah@milkyway.gsfc.nasa.gov)

This subroutine parses a string containing a delimited list into an array of strings, one element for each member of the list. The calling sequence is:

```
SUBROUTINE parse_list(in_str, out_array, in_delim)

CHARACTER(LEN=*), DIMENSION(:), POINTER :: out_array
CHARACTER(LEN=*),                       :: in_str
CHARACTER(LEN=1), OPTIONAL, INTENT(IN)  :: in_delim
```

where `in_str` is the string to be parsed, `out_array` is returned array of strings, and `in_delim` is the delimeter seperating the list items. `in_delim` is an optional parameter, and if not given defaults to a space. Note that in any case list items may not contain spaces. The user also must take care that the pointer passed as `out_array` has sufficient length to hold the parsed strings.

Additional, if `in_str` begins with an "@", then the string is taken as a filename the list items, one item for each line of the file.

23 A tool to regrid data from one 2D pixel grid to another

Module name: `polygon`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

24 Contains an analytic approximation to the off-axis PSF, and routines to sample it.

Module name: `psf_ims`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

25 Routines to return random numbers in various distributions

Module name: `random_aux`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

26 Array Reallocation

Module name: `reallocate`

Author: Dean Hinshaw (NASA/GFSC, dah@milkyway.gsfc.nasa.gov)

This subroutine can be used to reallocate memory space for a pointer to an array, retaining any data already stored in the array. The calling sequence is:

```
SUBROUTINE realloc_real32(p, n)

INTEGER, INTENT(in) :: n
```

where p can have any of the possible specifications:

```
REAL(KIND=single),   POINTER, DIMENSION(:) :: p
REAL(KIND=double),   POINTER, DIMENSION(:) :: p
INTEGER(KIND=int8),  POINTER, DIMENSION(:) :: p
INTEGER(KIND=int16), POINTER, DIMENSION(:) :: p
INTEGER(KIND=int32), POINTER, DIMENSION(:) :: p
LOGICAL(KIND=bool),  POINTER, DIMENSION(:) :: p
CHARACTER(LEN=*),    POINTER, DIMENSION(:) :: p
```

and n is the size of the reallocated array. The lower bound value of the old array is retained.

If n is greater than the original array size, then the all data from the old array is retained, and the array values greater then the original array size are undefined. If n is less than the original array size, then the first n data elements from the old array are retained.

27 Utilities to rebin 1D or 2D data between parallel pixel grids

Module name: rebinners

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

28 Utilities to regrid 1D or 2D data between parallel pixel grids

Module name: regridders

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)
Masaaki Sakano (University of Leicester, mas@star.le.ac.uk)

28.1 Calculates OldPixelCorners via an Affine transform

```
interface calcOldPixelCornersAffine
  subroutine calcOldPixelCornersAffineDouble(oldPixelCorners &
     , oldAryEdgesInfo, newAryEdgesInfo, mtrxLinTrans, vecTranslate)
    type(Point2dT), intent(out) :: oldPixelCorners(:,:)
    type(AryEdgesInfoT), intent(in) :: oldAryEdgesInfo, newAryEdgesInfo
    real(double),   intent(in)  :: mtrxLinTrans(2,2), vecTranslate(2)
  end subroutine calcOldPixelCornersAffineDouble

  subroutine calcOldPixelCornersAffineSingle(oldPixelCorners &
     , oldAryEdgesInfo, newAryEdgesInfo, mtrxLinTrans, vecTranslate)
    type(Point2dT), intent(out) :: oldPixelCorners(:,:)
```

```
    type(AryEdgesInfoT), intent(in) :: oldAryEdgesInfo, newAryEdgesInfo
    real(single),   intent(in)  :: mtrxLinTrans(2,2), vecTranslate(2)
  end subroutine calcOldPixelCornersAffineSingle
end interface
```

This subroutine gives an array `oldPixelCorners` as an argument to pass to `regridCartesian()`, when an Affine transformation (A and B) as given below is the coordinate transformation used in regridding,

$$\begin{pmatrix} x_{new} \\ y_{new} \end{pmatrix} = A * \begin{pmatrix} x_{old} \\ y_{old} \end{pmatrix} + B$$

$$= \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_{old} \\ y_{old} \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

where the matrix A (=mtrxLinTrans) is a component for the linear transformation and B (=vecTranslate) is for the translation (a.k.a. parallel move). Note

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \texttt{mtrxLinTrans(1,1)} & \texttt{mtrxLinTrans(1,2)} \\ \texttt{mtrxLinTrans(2,1)} & \texttt{mtrxLinTrans(2,2)} \end{pmatrix}$$

The returned `oldPixelCorners` can be directly passed to `regridCartesian()`.

Note `oldPixelCorners` can contain the values which are out of the boundary of `newAryEdgesInfo`, such as zero or negative values.

Among the input arguments, the size of `oldPixelCorners` has to be larger by 1 than those specified in `oldAryEdgesInfo%arySize`.

`type(Point2dT)` is defined in `geometric_types`. `type(AryEdgesInfoT)` is defined in `oldAryEdgesInfo` and `newAryEdgesInfo` is defined in `array_utils`.

As examples,
(A) When oldAry(1:5,1:5), newAry(1:5,1:5), identical transformation.

```
oldPixelCorners(1,1)\%x,y == (0.5, 0.5)
oldPixelCorners(6,6)\%x,y == (5.5, 5.5)
```

(B) When oldAry(1:5,1:5), newAry(-1:6,0:8), identical transformation.

```
oldPixelCorners(1,1)\%x,y == (2.5, 1.5)
oldPixelCorners(3,1)\%x,y == (4.5, 1.5)
```

because the respective indices of 3rd and 2nd for X and Y in newAry correspond to (1,1) in oldAry.

28.2 Regrids in the Cartesian coordinates

```
interface regridCartesian
```

```
  subroutine regridScalar(oldPixelCorners, oldImage, newImage, status&
  , testFlagArg, averagingStyle)
    real(single),   intent(in)            :: oldImage(:,:)
    type(Point2dT), intent(in)            :: oldPixelCorners(size(oldImage,1)+1&
                                                            ,size(oldImage,2)+1)
    real(single),   intent(out)           :: newImage(:,:)
    integer,        intent(out), optional :: status
    logical(bool),  intent(in),  optional :: testFlagArg
    character(*),   intent(in),  optional :: averagingStyle
  end subroutine regridScalar

  subroutine regridScalarDouble(oldPixelCorners, oldImage, newImage, status&
  , testFlagArg, averagingStyle)
    real(single),       intent(in)            :: oldImage(:,:)
    type(Point2dDbleT), intent(in)            :: oldPixelCorners(&
                                                   size(oldImage,1)+1,&
                                                   size(oldImage,2)+1)
    real(single),       intent(out)           :: newImage(:,:)
    integer,            intent(out), optional :: status
    logical(bool),      intent(in),  optional :: testFlagArg
    character(*),       intent(in),  optional :: averagingStyle
  end subroutine regridScalarDouble

  subroutine regridVector(oldPixelCorners, oldImages, newImages, status&
  , testFlagArg, averagingStyle)
    real(single),   intent(in)            :: oldImages(:,:,:)
    type(Point2dT), intent(in)            :: oldPixelCorners(&
                                               size(oldImages,2)+1,&
                                               size(oldImages,3)+1&
                                             )
    real(single),   intent(out)           :: newImages(:,:,:)
    integer,        intent(out), optional :: status
    logical(bool),  intent(in),  optional :: testFlagArg
    character(*),   intent(in),  optional :: averagingStyle
  end subroutine regridVector
end interface
```

This subroutine is intended to allow rebinning of an image from one cartesian coordinate system to another. Now, we define here an image as a two-dimensional array of uniform rectangular pixels. If we change the coordinate system, this image becomes distorted. It is nice to be able to convert it once again to an array of rectangular pixels, but this time in the new coordinate system. This involves taking each of the old, distorted pixels and dividing its contents up among the new pixels. It is assumed here that (i) the distorted pixel still has straight 'sides', ie it is a quadrilateral (a polygon for generality); (ii) that the value within the old pixel is evenly distributed through the pixel (this assumption *MAY BE INVALID* for extremely nonlinear distortions); (iii) that the distorted pixel is not folded over – ie that no two of its sides cross; (iv) that the old pixels are not greatly larger than the new - the present algorithm still works ok in such a regime, but the result will look 'steppy.' In this case an interpolation algorithm would yield smoother-looking results.

The relation between the old and new coordinate systems is here entirely contained within the input array oldPixelCorners. This gives the coordinates of each pixel in the array oldImage, expressed in the wcs 'pixel' system of newImage. What does 'wcs pixel system' mean? It means that the width and height of pixels in newImage are both equal to 1.0 and the centre of the pixel newImage(1,1) is at (1.0, 1.0).

NOTE `newImages` is **NOT** defined if any error is detected before processing.

Note `type(Point2dT)` is defined in `geometric_types`.

29 Short cuts to saving (or overwriting) output images

Module name: `save_image`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)
Masaaki Sakano (University of Leicester, mas@star.le.ac.uk)

The subroutines/functions in this module provide functions/subroutines to save a Fortran array as a FITS image.

29.1 saveDetImage(): Save DETX/DETY images

Subroutine to save a DETX/DETY image from a given 2-dimensional array.

```
interface saveImage
  subroutine saveDetImageDouble(detImage, detImageSetName \&
     , detImageEdgesInfo, templateFitsSetName, wcsExtended, detWcs \&
     , strTelescop, strInstrum)
    real(double), intent(in)           :: detImage(:,:)
    character(*), intent(in)           :: detImageSetName
    type(aryEdgesInfoT), intent(in), optional :: detImageEdgesInfo
    character(*), intent(in), optional :: templateFitsSetName, strTelescop, strInstrum
    type(WcsAxesExtendedT), intent(in), optional :: wcsExtended
    type(WcsT),   intent(in), optional :: detWcs
  end subroutine saveDetImageDouble

  subroutine saveDetImageSingle()
    real(single), intent(in)           :: detImage(:,:)
  end subroutine saveDetImageSingle
  subroutine saveDetImageInt32()
  end subroutine saveDetImageInt32
  subroutine saveDetImageInt16()
  end subroutine saveDetImageInt16
  subroutine saveDetImageInt8()
  end subroutine saveDetImageInt8
end interface
```

As for the input array (`detImage`), all the Real and Integer types are allowed, and that is the only difference in the interface.

`detImageSetName` is the output FITS filename. `clobber` is taken into account.

`detImageEdgesInfo` (optional) is the frame information of the input array. (The type is defined in `array_utils`). If not given, it is calculated via `getDetImageEdgesInfo()`.

`wcsExtended` can be given instead of, or in addition to, `detImageEdgesInfo` in order to directly control the coordinate information in the output header attributes. In that case, make sure

`wcsExtended%withPhysical == .true.`
if you want to add the PHYSICAL coordinates information in the output file. If both `wcsExtended` and `detWcs` are given, the WCS information is overwritten, where possible, according to `detWcs` at the end. Obviously `detWcs` can not include any PHYSICAL coordinate information.

If `templateFitsSetName` is given, all the primary header attributes except for those for DSS and WCS are copied to the output file.

TELESCOP attribute can be directly specified via `strTelescop`; otherwise, unless `templateFitsSetName` is given and has the attribute, the default '`XMM`' is written in the output file.

INSTRUME attribute can be directly specified via `strInstrum` (string), such as (EMOS1—EMOS1—EPN); in default this routine does nothing about it.

29.1.1 Examples

```
call saveDetImage(detImage, 'outimage.ds')
call saveDetImage(detImage, 'outimage.ds', templateFitsSetName='event.FIT')
call saveDetImage(detImage, 'outimage.ds', strInstrum='EMOS1')
```

29.2 getDetImageEdgesInfo(): Get a default frame information for a DET image

```
interface getDetImageEdgesInfo
  subroutine getDetImageEdgesInfoDouble(detImage, outBinSizeXY, outOffsetXY)
    real(double),  intent(in) :: detImage(:,:)
    real(single),  intent(in), dimension(2), optional :: outBinSizeXY, outOffsetXY

    type(aryEdgesInfoT) :: getDetImageEdgesInfo
  end subroutine getDetImageEdgesInfoDouble

  subroutine getDetImageEdgesInfoSingle()
  end subroutine getDetImageEdgesInfoSingle
  subroutine getDetImageEdgesInfoInt32()
  end subroutine getDetImageEdgesInfoInt32
  subroutine getDetImageEdgesInfoInt16()
  end subroutine getDetImageEdgesInfoInt16
  subroutine getDetImageEdgesInfoInt8()
  end subroutine getDetImageEdgesInfoInt8
end interface
```

As for the input array (`detImage`), all the Real and Integer types are allowed, and that is the only difference in the interface.

`outBinSizeXY` (optional) is a 1-dimensional array with the size of 2. The default is (/ 80.0, 80.0 /).

`outOffsetXY` (optional) is a 1-dimensional array with the size of 2; if `outOffsetXY=(/a, b/)` is given, the (a,b) in PHYSICAL coordinates is located at the centre of the array. In default, (a, b)==(0.0, 0.0).

This function returns `type(aryEdgesInfoT)` (defined in `array_utils`).

30 Quick Sorting

Module name: `sort_mod`

Author: Clive Page (University of Leicester, cgp@star.le.ac.uk)

This module contains subroutines to sort a data array into ascending order using Hoare's quick-sort algorithm. There is a generic interface which supports data types `INTEGER`, `REAL`, `DOUBLE PRECISION`, and `CHARACTER` (any length).

The simplest call is:

```
CALL quick_sort(array)
```

The array argument has `INTENT(INOUT)` and returns the data sorted into ascending order.

In some cases it is desirable to know the original order of the data points, for example to sort another array in the same way. In this case an optional second argument may be given; it returns an integer array of the same size containing numbers in the range 1 to size(array) which tell you the original position of each element returned sorted. For example if you do:

```
unsorted_array = array
call quick_sort(array, index)
```

then unsorted_array(index(i)) = array(i) for all i in [lbound(array), ubound(array)]. Note that array is always returned sorted, whether index is supplied or not. This can be something to be careful of. Suppose you have a data structure array which you want to sort in order of one of its constituents, for example a structure that stores gtis:

```
type :: gtiType
  real(kind(0d0)) :: time
  logical         :: isStart
end type gtiType
type(gtiType) :: gtiArray(100)

! Fill gtiArray
```

In this case to sort the logicals as well you will need to do something like the following:

```
type(gtiType) :: temp_gtiArray(size(gtiArray))

temp_gtiArray%time = gtiArray%time
call quick_sort(temp_gtiArray%time, index)
do i = 1, size(gtiArray)
  temp_gtiArray(i)%isStart = gtiArray(index(i))%isStart
! NOT temp_gtiArray(i) = gtiArray(index(i))!! The times are already sorted.
end do
gtiArray = temp_gtiArray
```

Note that the quick-sort algorithm is on average about twice as fast as heap-sort but becomes much slower for special cases. This quick-sort algorithm was designed to cope with nearly-sorted data as well as random data without any significant degradation in speed. Note that it is *not* a stable sort, i.e. equal values will not necessarily remain in the same relative order.

31 A routine which returns circles or ellipses to mark source locations

Module name: `source_cutouts`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

32 1D and 2D cubic-spline routines

Module name: `splines`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

33 Miscellaneous utilities

Module name: `ssc_misc`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk) (except for getFreeIoUnit)

33.1 Find a Free I/O Unit

Author: Clive Page (University of Leicester, cgp@star.le.ac.uk)

This subroutine returns a number of a free I/O unit, i.e. one that is not currently allocated to and file. The calling sequence is:

33.2 stripStr()

```
subroutine stripStr(inStr, outStr, isPreceedingOnly)
  character(*), intent(in)  :: inStr
  character(*), intent(out) :: outStr
  logical, intent(in), optional :: isPreceedingOnly ! .false. in default.
end subroutine stripStr
```

This "strips" the input string, namely removes the preceeding and trailing spaces, tabs, line-feeds, carriage-returns. If `isPreceedingOnly` is given and TRUE, no trailing space is deleted.

33.3 splitStr()

```
subroutine splitStr(inStr, outStrAry)
  character(*), intent(in)  :: inStr
  character(*), pointer :: outStrAry(:) ! intent(out)
end subroutine splitStr
```

This "splits" the given string with the delimeter of consecutive spaces into an array, and returns it as the pointer character array.

`outStrAry` should not be initialised before the call. **NOTE** make sure to deallocate `outStrAry(:)` after the call.

34 Some functions for testing/debugging

Module name: `test_utils`

Author: Masaaki Sakano (University of Leicester, mas@star.le.ac.uk)

The subroutines/functions in this module provide functions that are useful in testing (and possibly debugging).

34.1 isNearlyEqual(): Comparing numbers with a given precision

This function returns `.true.` if the given pair of values agree with each other in the given precision (order), or `.false.` otherwise.

```
LOGICAL(bool) FUNCTION isNearlyEqual(cmp, compared, precision)
  REAL,    intent(in) :: cmp, compared   ! or INTEGER
  INTEGER, intent(in) :: precision
END FUNCTION isNearlyEqual
```

For the pair of the first two arguments, any combination of `int8`, `int16`, `int32`, `single` and `double` is allowed.

34.1.1 Examples

`isNearlyEqual(1110, 1112, 3)` returns `.true.`
`isNearlyEqual(1.2, 1.0, 3)` returns `.false.`
`isNearlyEqual(1.234, 1.231, 3)` returns `.true.`
`isNearlyEqual(1.2349, 1.2351, 3)` returns `.true.`

35 Routines to perform hyperbolic distortion of values in the interval [0:1]

Module name: `warp_utils`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

36 Utilities to assist in the reading and manipulation of WCS keywords

Module name: `wcs_aux`

Author: Ian Stewart (University of Leicester, ims@star.le.ac.uk)

37 Utilities to assist development in Perl

Module name: `SSCLib`

Author: Masaaki Sakano (University of Leicester, mas@star.le.ac.uk)

See the header of the library code for detail. You may want to read it by, for example, `cd /YOUR/DIR; pod2man SSCLib.pm | tbl | neqn | nroff -h -man | less`

38 General coordinates class in Perl

Module name: `Coords`

Author: Masaaki Sakano (University of Leicester, mas@star.le.ac.uk)

See the header of the library code for detail. You may want to read it by, for example, `cd /YOUR/DIR; pod2man Coords.pm | tbl | neqn | nroff -h -man | less`

39 Celestial coordinates utilities in Perl

Module name: `CelCoords`

Author: Masaaki Sakano (University of Leicester, mas@star.le.ac.uk)

See the header of the library code for detail. You may want to read it by, for example, `cd /YOUR/DIR; pod2man CelCoords.pm | tbl | neqn | nroff -h -man | less`

References