Part II. Data Processing

Contents

1 MASS Software. Part II: Data Processing

1.1 Instroduction

This document represents the second part of the MASS Software Reference Manual and gives a detailed information on how the scientific output information is produced from the PMT count data. The preceding Part I dealt with the MASS device control issue where the place of scientific modules in the general hierarchy of TURBINA modules was specified. So, the question "where, when and how the [atmos/scind/scan] module is used in TURBINA" should be addressed there.

1.2 Scientific modules for MASS

There is a number of modules which serve for *handling* of scintillation indices (SI), atmospheric parameters and turbulence profiles etc. "Handling" means the creation/calculation of respective entities, their write/read operations with files and export in other modules via some so called "get"-functions.

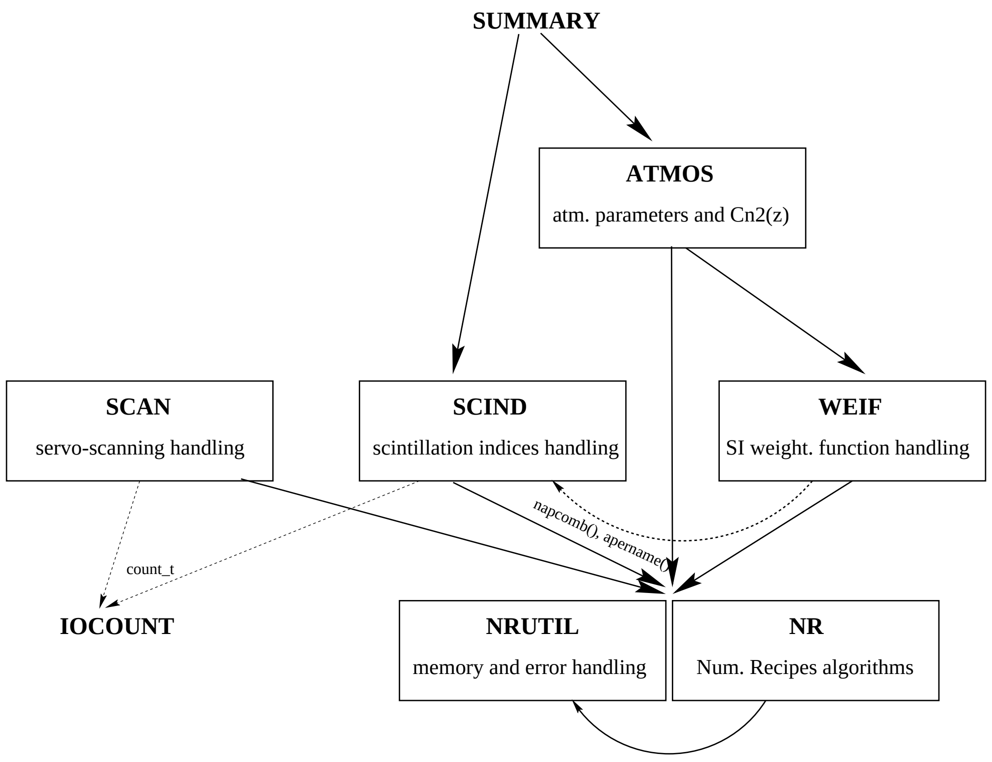

Figure 1: Organization of scientific modules in MASS Software

Scientific modules are organized in three hierarchical levels. On the top, the atmospheric calculations module stays (ATMOS), on the bottom - the memory and mathematics (NR and NRUTIL). In the middle - the count-processing utilities SCIND and SCAN and theoretic weight calculator WEIF.

The module on a particular level can refer and use the modules from the same level and below which is reflected by the arrows on the figure. The module names without boxes stand for the ones outside the "scientific" scope of modules for TURBINA (e.g. the night-report creator SUMMARY). Dashed arrows represent the *weak* dependence and the particular structures (functions or types) which are taken from the module for usage are indicated near such arrows. For example, it's only the type count_t of the detector pulse counts which is taken from the (non-scientific) module IOCOUNT.

Each module has a header section names "Usage" which describes the sequence of calls to particular procedures which maintains the desired way of data processing.

See the **File Index** for the further detailed description of these modules.

2 Part II. Namespace Index

2.1 Part II. Namespace List

Here is a list of all documented namespaces with brief descriptions:

3 Part II. Compound Index

3.1 Part II. Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

4 Part II. File Index

4.1 Part II. File List

Here is a list of all documented files with brief descriptions:

5 Part II. Namespace Documentation

5.1 atm Namespace Reference

Namespace atm contains the set of functions of highest logical level for atmospheric calculations.

Enumerations

- enum **ordint** { **FSEE**, **WSEE**, **FM0**, **WM0**, **FHEFF**, **WHEFF**, **ISOPL**, **M2**, **TC** }
- enum **what** { **ATMPAR**, **CN2PROF**, **O_CSCIND** }
- enum **poweridx** { **IFSE**, **IWSE**, **IISP**, **IEFF**, **IISK** }
- enum **cn2method** { **FIXEDLAY** = 'X', **FLOATLAY** = 'L' }

Functions

- void **update** (const char ∗wfile, const char ∗checkfile, double zshift1, double zshift2, double z0=**wf::Z0**, double zmax=**wf::ZMAX**, double dz=**wf::DZ**, double dzmin=**wf::DZMIN**)

  Update the data structures for atmospheric calculations with a new weight function file.

- void **done** ()

  Deallocate the memory reserved for module structures.

- double **getval** (**ordint what**)

  Return calibrated atmospheric parameter.

- double **geterr** (**ordint what**)

  Return the **relative** *error of the calibrated atmospheric parameter.*

- double **getcn2see** ()

  Return the seeing computed from the last restored profile.

- int **getncn2** ()

  Return number of the Turbulence profile altitudes.

- double **getzcn2** (int i)

  Get the altitude of the restored turbulence profile layer.

- double **getcn2** (int i)

  Get the strength of the restored turbulence profile layer or chi-square.

- void **calcint** (double ∗scind, double ∗scinds)

  Compute the atmosphere integral parameters and Cn2-profile.

- double **calcn2** (double ∗scind, double ∗scinds, double ∗e2scind, double ∗e2scinds, double zdist, char meth)

  Restore the low-resolution Cn2 profile.

- const double ∗ **geto_csi** ()

  Get the normalized observed-model indices restored by **calcn2***()* (p. 13).

- void **avgint** (double desi, double edesi, double zdist)

  Average Cn2 integrals and compute from them the atmospheric parameters.

- void **write** (FILE ∗f, const char ∗stamp, **atm::what what**, bool header=false)

  Write computed atmospheric parameters or restored Cn2 profile or the header line for one of them.

- void **interpolate** (double ∗z, double ∗intcn2, int nz, double ∗zin=0, double ∗cn2in=0, int nzin=0)

  Interpolate the **last** *restored Cn2 profile into new grid.*

- const char ∗ **getfmt** (**atm::ordint** i)

  Get format respective to integral ID.

- const char ∗ **getname** (**atm::ordint** i)

  Give a name of a field in output file.

Variables

- const bool **HEAD** = true
- const int **NMODE** = 2
- const double **MINRELW** = 1e-3
- const int **MAXNBASE** = 10
- const int **NMEAS** = 256
- const double **POWSEE** = 0
- const double **POWISP** = 5/3.
- const double **POWEFF** = 1.0
- const double **POWISK** = 2.0
- const int **NPOWER** = 5
- const char **USEWEIGHT** [**NPOWER**][2 *10+1]
- const char **FITSCIND** [ ]
- const double **HBOUND** = 1.0
- const double **KS** = 1.73588e+07
- const double **KP** = 0.000756348
- const double **KT** = 0.175
- const double **DZCN2** = 1
- const char **ERRFMT** [ ] = "%s%7.3f"
- const char **ANGFMT** [ ] = "%s%5.2f"
- const char **HEFFMT** [ ] = "%s%5.0f"
- const char **TAUFMT** [ ] = "%s%5.2f"
- const char **ATMDLM** [ ] = " "
- const char **CHI2FMT** [ ] = "%s%6.2f"
- const char **ALTFMT** [ ] = "%s%4.1f"
- const char **CN2FMT** [ ] = "%s%8.2e"

5.1.1 Enumeration Type Documentation

5.1.1.1 enum atm::ordint

Order of atmosphere integrals in their storage matrix and output file line. In a file, each value is followed by its error. The element "LAST" must always stay at the end since it gives a number of integrals in enumeration

Enumeration values:

FSEE	free seeing
WSEE	whole atm. seeing
FM0	non-converted integral of Cn2 over free atmosphere
WM0	non-converted integral of Cn2 over whole atmosphere

FHEFF effective altitude of turbulence in free atm.

WHEFF effective altitude of turbulence in whole atm.

ISOPL isoplanatic angle

M2 second moment of turbulence (integral(Cn2(h)∗h^2dh))

TC atmospheric time constant

5.1.1.2 enum atm::what

Writing selector in **write**() (p. 17):

Enumeration values:

ATMPAR write atmospheric parameters

CN2PROF write the restored Cn2 profile with altitudes and Chi-square

O_CSCIND write normalized observed-model scintillation indices

5.1.1.3 enum atm::poweridx

Arrangement of the moment decomposition coefficients in the coef[] array

Enumeration values:

IFSE coef[Free seeing][]:

IWSE coef[Seeing][]

IISP coef[Isoplanatic angle][]

IEFF coef[Turbulence effective altitude][]

IISK coef[Isokinetic angle][]

5.1.1.4 enum atm::cn2method

Enumeration values:

FIXEDLAY "fiXed" layers method

FLOATLAY "fLoating" layers method

5.1.2 Function Documentation

5.1.2.1 void update (const char ∗ *wfile*, const char ∗ *checkfile*, double *zshift1*, double *zshift2*, double *z0* = wf::Z0, double *zmax* = wf::ZMAX, double *dz* = wf::DZ, double *dzmin* = wf::DZMIN)

Parameters:

wfile The file name of the weight matrix (see **wf_t::write**() (p. 71))

checkfile file to print the moment approximation information for checking (may be null)

zshift1 Altitude shift (in [km], normally **non-positive)** for the normal mode or for *one* part of the generalized mode measurements

zshift2 Altitude shift (in [km], normally **negative)** for the *rest* part of the generalized mode measurements

z0 Cn2 moments decomposition altitude grid: see **wf_t::setzgrid**() (p. 70)

zmax see **wf_t::setzgrid**() (p. 70)

dz see **wf_t::setzgrid**() (p. 70)

dzmin see **wf_t::setzgrid**() (p. 70)

This function should be called each time the scintillation weight functions are changed due to some reason (new star has another spectrum, new system magnification etc.) or the measurement mode parameters (altitude shift in generalized mode, number of Base-times per Accumulation time, etc.) are changed.

First, if the supplied name of the weight function file differs from that loaded before (the name is stored internally), this new weight functions set is read from the file. Also, from the weight file the number of indices (equal to that of weights) is derived.

If neither the weight functions file name, nor the altitude z-shifts of the pupil are changed, nothing is done but only the reset of the computed atmospheric parameters.

If either the weights or z-shifts are changed, then the calculations of the Cn2 moments are made. They involve the shift of the weights set by *zshift1* and *zshift2*. These shifts can be computed with **wf::getzshift**() (p. 47) and are returned negative by this function for positive focal lengths supplied. Thus, the negative zshift implements the generalized mode.

The decomposition also involves the approximation of the power-law dependences of altitude by a set of these shifted weight functions. The SVD method is used for this. Approximation enables us to represent the Cn2-moments by a set of scintillation indices. If *zshift1* or *zshift2* differ from each other, then both the coefficients for generalized and non-shifted modes are obtained. The coefficients are stored in global arrays to be used in **calcint**() (p. 12).

The computations are done in a loop which is passed twice.

In a **first** loop, the job is done for the Normal mode measurements with one altitude shift equal to *zshift1* (normally it is 0). It goes through steps:

- the set of weight functions is shifted by *zshift1* and interpolated into the equidistant grid with 1km spacing. The obtained matrix of weights is saved in a global 2-DIM array for usage in **calcn2**() (p. 13).

- the set of weight functions is again shifted by *zshift1* and interpolated into the *user-defined* altitude grid set by *z0*, *zmax*, *dz*, *dzmin.*

- the altitude grid is scanned and the grid index IBOUND where the altitude reaches the boundary layer height (∼1km) is found.

- the shifted weight matrix is copied with *transposition* into the matrix **W** for SVD decomposition. It has dimensions *ncol=Nw* x *nrow=Nz*, where the number of weights *Nw* is less than the number of altitudes *Nz.* This is ensured in advanced in init().

- the obtained matrix **W** is SVD-decomposed by NR utility **nr::svdcmp**() into two *orthogonal* matrices **U** and transposed **V** and a diagonal matrix of *singular* values *diag(w)*:

$$(\mathbf{W}) = (\mathbf{U}) \times diag(w_1 ... w_{Nw}) \times (\mathbf{V^T})$$

- the small singular values are rejected (reset to 0) to have the minimal ratio w_i/w_{max} more than **atm::MINRELW** (p. 19). If there are still more than **atm::MAXNBASE** (p. 19) singulars, the least values are also rejected.

- the altitude power vectors *h=\em* z^k are produced for k=0, 1, 5/3 and 2. The zero power (for seeing related Cn2-integral) is made two ways - normal exact unity for whole-atmosphere seeing and with "dumped" values below the altitude grid index IBOUND for "free atmosphere" seeing. The latter is equal to 1 above the boundary layer and mimics the AB-channels differential scintillation index weight below 1km.

- for each altitude power, the *back-substitution* utility **nr::svbksb**() is used to produce the coefficients of altitude powers decomposition by the weight functions presented in **W**:

$$\mathbf{coef} = (\mathbf{V}) \times diag(1/w_1 ... 1/w_{Nw}) \times (\mathbf{U^T} \cdot \mathbf{h})$$

  These coefficients **coef** - vectors[*Nw*] - are stored as lines of a matrix with rows number equal to the number of decomposed powers of altitude (5).

- the quality of the approximation of the altitude moments with weight functions may be checked if one supplies the non-null file name *checkfile*. Then utility writes then the decomposition coefficients and the approximated relations in this file. The file *checkfile* is *rewritten* each time (not appended), and consists of two similar parts - the first for approximations with *zshift1* -shifted weights (normally - non-shifted) and the second for approximation with both *zshift1* and *zshift2* -shifted weights.

In a **second** loop, the similar work is done using the second altitude shift *zshift2*. Thus, the *shifted* weights which are marked as usable in atm::USESHFT are **added** in the weight matrix for the Cn2 profile restoration and in the matrix **W** (rightmost columns this time) for the decomposition of altitude powers. The dimension *Nw* is increased compared to the first loop by the number of used *shifted* weights.

The derived decomposition coefficients are saved as another matrix of **coef** vectors. In the module, the pointers to these two matrices obtained in two loops are stored as a 2-element vector of pointers. First element (i.e. matrix) is accessed by the Normal mode results processing, the second is accessed by the Generalized mode processing.

Before exit, the atmospheric integral converted parameters, Cn2 profile moments and their errors and number of layers (returned by **getncn2**() (p. 11)) are reset to zero.

No job is done if error is set (**nr::erget**() (p. 81)).

5.1.2.2 void done ()

Returns:
void

This function releases the memory allocated for weights, SED, coefficients of Cn2-moments decomposition etc by init().

Note:
This function must be kept in strict sync with init(), **update**() (p. 7) and alloc().

5.1.2.3 double getval (ordint *what*)

Parameters:
what one of computed parameters - seeing, Heff, isoplanatic angle etc. presented in ordint enumeration

Returns:
value or 0 on error

See also:
avgint() (p. 15)

5.1.2.4 double geterr (ordint *what*)

Parameters:
what one of computed parameters - seeing, Heff, isoplanatic angle etc. presented in ordint enumeration

Returns:
relative error value or "bad" value on error

If the value of parameters is not "bad", the ratio of the error of the parameter to the parameter value itself is returned; otherwise, zero is returned

See also:
avgint() (p. 15)

5.1.2.5 double getcn2see ()

Returns:
Seeing by Cn2[], arcsec

5.1.2.6 int getncn2 ()

Returns:
Length of the private computed Cn2-array

Note that this number is not necessary the length of allocated array *cn2[]*.

See also:
calcn2() (p. 13)

5.1.2.7 double getzcn2 (int *i*)

Parameters:

i layer index [0..**getncn2**() (p. 11)-1]

Returns:

altitude [km]

See also:

calcn2() (p. 13)

5.1.2.8 double getcn2 (int *i*)

Parameters:

i layer index [0..**getncn2**() (p. 11)-1] or **getncn2**() (p. 11) for Chi-square

Returns:

strength [m^-2/3]

Last element of required array contains the Chi-square measure of the fitted profile accessible as getcn2(**getncn2**() (p. 11)).

See also:

calcn2() (p. 13)

5.1.2.9 void calcint (double * *scind*, double * *scinds*)

Parameters:

scind array of non-shifted indices

scinds array of shifted indices (may be NULL)

Using the Cn2-moments decomposition coefficients and scintillation indices, the calculation of integral atmospheric parameters is performed. Number of indices in either *scind* or *scinds* must be equal to number of weights determined in **update**() (p. 7) call. No check of the validity of supplied *scind(s)* arrays is done.

The coefficients of multiplication of the indices in normal and generalized modes to obtain the turbulence moments are already precomputed in **update**() (p. 7); the results become accessible after the function call by **getval**() (p. 10) and **geterr**() (p. 11). The atmospheric time constant is computed given the *desi* parameter. If *scind(s)* or *desi* parameters are empty

(NULL or 0, respectively), the calculations for supplied not-NULL parameters are only made.

The turbulence moments of the power k (see power list in **update**() (p. 7)) are computed easily as a scalar product:

$$M_k = (\overline{\mathbf{s}} \cdot \mathbf{coef}_k)$$

Here the vector **s** of scintillation indices is composed of all non-shifted normal and differential indices and shifted indices marked by '1' in atm::USESHFT.

The computed quantities are saved in the current line of the global storage for further averaging by **avgint**() (p. 15). This storage is enlarged automatically once it is filled (initial size is **atm::NMEAS** (p. 19)).

Performance

One calculation of integrals takes **5** ms at PC P-III 667 MHz.

5.1.2.10 double calcn2 (double ∗ *scind*, double ∗ *scinds*, double ∗ *e2scind*, double ∗ *e2scinds*, double *zdist*, char *meth*)

Parameters:

scind array of average non-shifted indices

scinds array of average shifted indices (may be NULL)

e2scind array of *squared* errors of *scind*

e2scinds array of *squared* errors of *scinds*

meth atm::FIXED: Use fixed-altitude layers restoration, atm::FLOAT: search for four strongest layers

zdist zenith distance in [degree]

Returns:

Minimal chi-square reached, or "bad" value on error

The profile restoration results in altitudes of the **getncn2**() (p. 11) layers altitudes **getzcn2**() (p. 12) with strengths **getcn2**() (p. 12). Two methods are implemented - the search for four strongest layers (if not *isfixed)* and the search for the intensity of six layers placed proportionally in the altitude range with relative resolution 0.5.

Similarly to **calcint**() (p. 12), the vector **s** of scintillation indices is composed of all non-shifted normal and differential indices and shifted indices marked by '1' in **atm::USEWEIGHT** (p. 19). The length of vector is equal (by definition) to the number of used weights *Nw*.

Both methods operate currently with only integer-number altitudes (i.e. 0,1,2...km). Meanwhile, the values of weight functions used for restoration are taken not at these altitudes but at those corrected by secans(zenith distance) factor.

Fixed layers method

Fixed layers are placed proportionally in altitude: 1, 2, 4, 8, 16 km. In the generalized mode when shifted indices *scinds* are available, the ground layer at 0km is added. Thus, *Nlay=5* or 6 layers are fixed in the Cn2 profile altitude grid *z[]* (see **getzcn2**() (p. 12)). The 5 or 6 free parameters are thus these layer strengths.

The NR utility **nr::powell**() is used to minimize the merit function of the argument vector *y[Nlay]* used to produce the *synthetic* scintillation indices $\overline{s^2(syn)}$

$$\chi^2 = \sum_{i=0}^{Nw} \frac{(s^2(syn)_i - s^2(obs)_i)^2}{\varepsilon^2_{s^2(obs)_i}} \quad , \text{ where } \quad s^2(syn) = \sum_{i=0}^{Nlay} y_i^2 \cdot W(z_i)$$

Here the weight **W** is taken from the matrix created for the profile restoration by **update**() (p. 7). The argument y of the merit function is taken quadratically to set the non-negativity constraint on the restored profile strengths. Best fitted *y-s* are simply squared to produce Cn2-s (see **getcn2**() (p. 12)).

Floating layers method

The three trial layers are placed in all the possible combinations in a grid with 1km spacing between the lowest fourth (fixed) layer and the maximal altitude *zmax* (see init()). Minimal distance between layers is MINDH=1km. Number of layers is thus *Nlay=4*; number of free parameters is 3+4=7. For all trial combinations of layer altitudes, the *direct* method of the restoration of the layers' strengths is performed.

For this direct method, the weight matrix *W[Nlay* x Nw] is composed from the combined weight matrix produced in **update**() (p. 7) by selecting the needed columns corresponding to trial altitudes.

This matrix **W** is inversed with the help of SVD method **nr::svdcmp**() (see **update**() (p. 7))

$$(\mathbf{W})^{-1} = (\mathbf{V}) \cdot (diag(1/w)) \cdot (\mathbf{U}^T)$$

Then the strengths corresponding to the trial layers are directly computed:

$$\overline{C_n^2(syn)} = \overline{\mathbf{s_{obs}}} \cdot (\mathbf{W})^{-1}$$

Using these strengths instead of y^2 in a merit function formulae (see above), the chi-square quality of the restored profile with current trial altitudes is computed. The combination of altitudes when this chi-square is minimal is accepted and available by **getzcn2**() (p. 12) and **getcn2**() (p. 12).

In both methods, the resulting strengths of layers are divided by secans(zenith distance), to get the zenith-referenced result. Thus, the non-zenith position is accounted for twice, in altitudes (see beginning of description) and strengths.

Performance

One restoration takes typically **45** *ms* at PC P-III 667 Mhz for the method with 6 *fixed* layers, and **140** *ms* for placing 3 *floating* altitude layers plus one fixed.

No job is done if error is set (**nr::erget**() (p. 81)).

5.1.2.11 const double* geto_csi ()

Returns:

Pointer to residual scint.indices array where first Nidx elements are related to non-shifted indices and further go shifted indices (in generalized mode)

This function returns the relative deviations of observed indices from restored indices (normalized by observed indices) which correspond to just restrored Cn2 profile. The deviations are located in a placeholder, which is pointed by this function and which is deallocated by **done**() (p. 10). Do not deallocate thus!

5.1.2.12 void avgint (double *desi*, double *edesi*, double *zdist*)

Parameters:

desi average DESI for shifted channel A

edesi relative error of *desi*

zdist zenith distance in [degree]

Given the instantaneous integrals in their storage, their average values and standard deviations are computed and put in respective global vectors. The function **sc::avgmatrix**() (p. 33) is used for averaging. Errors thus take into account the correlation of values.

Then the average integrals are converted into atmospheric parameters - seeing, isoplanatic angle and effective altitude of turbulence. They are available via **getval**() (p. 10) and - relative errors - via **geterr**() (p. 11). Also, the atmospheric time constant is computed from provided *desi* and *edesi* values.

Seeing is computed as

$$\beta = KS\ \lambda^{-1/5} M_0^{3/5}$$

where the zero-moment Mo of the turbulence is:

$$M_0 = \int C_n^2(h)dh$$

(integration goes from 0 for whole-atm. seeing and starting from **HBOUND** for free-atm. seeing) and **KS** is the calibration constant.

Isoplanatic angle is computed as

$$\theta_0 = KP\lambda^{6/5} M_{5/3}^{-3/5} \sec^{8/5}(z)$$

where the 5/3-th moment of the turbulence profile is:

$$M_{3/5} = \int C_n^2(h) h^{5/3} dh$$

and $KP = (2.91\pi^2)^{-3/5}$ is the calibration constant.

Second moment of the turbulence is related to the differential astrometry precision and the operation of optical interferometers. Unlikely the isoplanatic angle, it is saved in files with no any calibration to any "angle".

The **free** atmosphere **effective altitude** is computed as

$$z_{eff}(free) = (M_{k_1}/M_{k_2})^{1/(k_1-k_2)}$$

where {*k1*} is 5/3 and {*k2*} is 1.

The value of Heff in the **whole** atmosphere is computed by definition:

$$z_{eff}(whole) = M_1/M_0$$

but only if the shifted indices are available.

The related to adaptive optics **atmospheric time constant** is computed from Differential Exposure Scintillation Index (DESI) σ_{DE}^2 as

$$\tau_{de} = KT \left[\sigma_{DE}^2\right]^{-3/5}$$

where **KT** is the empirical calibration constant.

All the used integrals are corrected for non-zero position of the star before conversion into atmospheric parameters. For this, they are divided by the factor $sec(\gamma)^{k+1}$, where γ is the zenith distance *zdist*.

Note:

During the conversion of integrals into atmospheric parameters, the integrals are checked to be positive. If at least one of integrals is invalid, the error **nr::ERNOD** (p. 84) is set which will block the writing of results by **write**() (p. 17) unless error is reset by **nr::erreset**() (p. 83) after **avgint**() (p. 15) and before **write**() (p. 17).

No job is done if error is set (**nr::erget**() (p. 81)).

5.1.2.13 void write (FILE ∗ *f*, const char ∗ *stamp*, atm::what *what*, bool *header* = false)

Parameters:

f FILE pointer

stamp leading word in the output line

what **atm::ATMPAR** (p. 7): write atm. parameters, **atm::CN2PROF** (p. 7): write the Cn2 profile

header Write the header line instead of data

This utility writes the output results of atmospheric calculations on the disk.

For **what**==*ATMPAR*, the single line is written on disk, consisting of the average atmospheric parameters each followed by its error. Parameters are free seeing ["], seeing ["], isoplanatic angle ["], second moment of turbulence [m^{4/3}], and atmospheric time constant [ms].

For **what**==*CN2PROF*, the Cn2-profile data line is written. It consists of 1) the number of adjusted Cn2 layers, 2) Chi-square 3) the **getncn2**() (p. 11) altitudes of layers and each followed by it strength in [m^-2/3]. 4) the seeing computed from the profile strengths

If *header==true*, the header line for respective **what** -selection is written instead of data. *stamp* must then contain the name of data which are normally written as a stamp when *header==false.*

No job is done if error is set (**nr::erget**() (p. 81)).

5.1.2.14 void interpolate (double ∗ *z*, double ∗ *intcn2*, int *nz*, double ∗ *zin* = 0, double ∗ *cn2in* = 0, int *nzin* = 0)

Parameters:

z the grid of altitudes to interpolate to [km]

intcn2 output array[nz] of interpolated cn2 values

nz length of *z[]*

zin the grid of input altitudes (last computed profile by default)

cn2in the grid of input Cn2 strength (last computed profile by default)

nzin the length of *zin[]* and *cn2in* if not default

The system response ("PSF") for the thin turbulence influence is assumed to be a Gaussian in a logarithm altitude domain with a 0.5 resolution. In case of two "close" layers (e.g. fixed layers method with altitude modification factor equal to 2), the linear interpolation is used. In other cases, the influence of two adjacent PSFs are accounted for or of the one when all the layers are above or below the current altitude *z*.

It is suggested to use this function with a proportional altitude grid *z[]* (in a logarithmic altitude domain) for the graphic representation of the Cn2 profile. The output array *intcn2[]* must be allocated before the function call.

The default parameters are aimed to replace the last-computed Cn2 profile data with other arrays *zin*, *cn2in*, both of the length *nzin*. This is needed when re-drawing all the profile data on the display.

5.1.2.15 const char∗ getfmt (atm::ordint *i*) `[inline]`

Parameters:

i Integral ID

Returns:

C-format

5.1.2.16 const char∗ getname (atm::ordint *i*) `[inline]`

Parameters:

i integral sequential number **atm::ordint** (p. 6)

The names atm::SEENAME ... atm::TAUNAME are returned; for "whole atmosphere" parameters - starting from the second character (first is "f" (free)).

5.1.3 Variable Documentation

5.1.3.1 const bool atm::HEAD = true

Last (defaulted to "write data, not header", i.e. "false") parameter in **write**() (p. 17)

5.1.3.2 const int atm::NMODE = 2

Number of modes: normal and generalized, hence = 2

5.1.3.3 const double atm::MINRELW = 1e-3

Minimal ratio of the singular value to the maximal one

5.1.3.4 const int atm::MAXNBASE = 10

Maximal number of orthonormal functions to use in h-powers restoration: (to apply after MINRELW based rejection)

5.1.3.5 const int atm::NMEAS = 256

Starting capacity of the integrals storage, measured in base-times per accumulation time. After filling the storage, it is reallocated with NMEAS-more capacity

5.1.3.6 const double atm::POWSEE = 0

Power of Cn2 moment for seeing

5.1.3.7 const double atm::POWISP = 5/3.

Power of Cn2 moment for isoplanatic angle

5.1.3.8 const double atm::POWEFF = 1.0

Power of Cn2 moment for Effective altitude

5.1.3.9 const double atm::POWISK = 2.0

Cn2 second moment power

5.1.3.10 const int atm::NPOWER = 5

Number of power laws to fit: 2 kinds of 0-th, 5/3-th, 1-st, 2-nd

5.1.3.11 const char atm::USEWEIGHT[NPOWER][2*10+1]

Initial value:

```
{
              "11001000000000000000",
        "10001000001000100000",
        "11111111111100100000",
        "11111111111100100000",
        "11111111111100100000" }
```

Binary mask for using the weights: "1"="use it". Each line consists of the normal weights mask and shifted weights mask concatenated. Now we thus use all non-shifted and As, Bs and ABs weights for all powers.

5.1.3.12 const char atm::FITSCIND[ ]

Initial value:

```
        "11111111111100100000"
```

Binary mask for fitting the particular indices with Cn2 model. As a rule, all the non-shifted and two least size shifted indices must be satisfied.

5.1.3.13 const double atm::HBOUND = 1.0

Height of the boundary layer [kilometers] to fit the Cn2-integral for free seeing (h^0 in integral is forced to resemble the AB-pair weight function below this height)

5.1.3.14 const double atm::KS = 1.73588e+07

Calibration constant in seeing formula (AstL 45, p.399): $seeing[''] = KS * \lambda_{eff}^{-1/5} M_0^{3/5} * \sec^{-1}(z)$, where *lambda_eff* is in [mkm]

5.1.3.15 const double atm::KP = 0.000756348

Calibration constant in isoplanatic angle formula (AstL 45, p.399) $\theta_0[''] = KP * (M_{5/3}/\lambda_{eff}^2)^{-3/5} * \sec^{8/5}(z)$, where $KP = (2.905 * (2\pi)^2 * (MKM2M)^{-2})^{-3/5} * 206265$

5.1.3.16 const double atm::KT = 0.175

Atmospheric time constant empirical calibration constant for DESI for 1,3ms integrations in diameter=2cm apertures with -1km shift

5.1.3.17 const double atm::DZCN2 = 1

Cn2-profile restoration weight matrix altitude spacing: 1km (non-fractional!)

5.1.3.18 const char atm::ERRFMT[] = "%s%7.3f"

The relative error format (same as in **scind.hxx**) with a leading "%s" for a delimiter (length corresponds to the length of longest accuracy heading)

5.1.3.19 const char atm::ANGFMT[] = "%s%5.2f"

Seeing and Isoplanatic angle [arcsec] format (with leading delimiter's %s)

5.1.3.20 const char atm::HEFFMT[] = "%s%5.0f"

Effective altitude [m] format (with leading delimiter's %s)

5.1.3.21 const char atm::TAUFMT[] = "%s%5.2f"

Atmospheric time constant format (with leading delimiter's %s)

5.1.3.22 const char atm::ATMDLM[] = " "

Delimiter of entries in the atmospheric data output file

5.1.3.23 const char atm::CHI2FMT[] = "%s%6.2f"

Chi-square writing format (with leading delimiter's %s)

5.1.3.24 const char atm::ALTFMT[] = "%s%4.1f"

Altitude of layers record format with a leading "%s" for a delimiter

5.1.3.25 const char atm::CN2FMT[] = "%s%8.2e"

Cn2-Profile values and second turbulence moment format (with leading delimiter's %s)

5.2 sc Namespace Reference

Declarations of all functions and constants to handle scintillation indices.

Enumerations

- enum **scwhat** { **DSI**, **DESI**, **AFLUX**, **AFLUXP**, **MOM** }

Functions

- int **napcomb** (int a)
- const char ∗ **apername** (int nchan, int index)

  Return the name for a given index of entrance aperture or combination of apertures.

- int **ind_seqnum** (int nchan, const char ∗name)

  Return the sequential number of index in storage by its name (ID).

- void **init** (int maxndata, int maxnmeas, int nchan)

  Initialize the structures to put the scintillation indices.

- void **chan_init** (int chan, double bkgr, double deadtime=-1, double nonpois=-1, count_t ∗data=0)

  Initialize channel data in static channel array element.

- void **compute** (int ndata, int ∗ncorr, double microexp, bool isgen)

  Compute the normal and differential (aperture and exposure) scintillation indices.

- int **getcurmeas** ()

  Get the number of measurements done in a current accumulation time.

- void **again** ()

  Reset the counter of accumulated indices in local index storages.

- void **done** ()

  Deallocate the memory reserved for module index storages.

- void **write** (FILE ∗f, **scwhat** what, const char ∗stamp, const char ∗suffix=0, bool header=false)

  Write the latest instant indices or count moments to the disk.

- void **avgmatrix** (double ∗∗matrix, int ncol, int nrow, signed char ∗rowselect, signed char select, double ∗avg, double ∗err2, int lag)

  Average a matrix of values along the second dimension with row selection.

- void **average** (bool isgen, int lag)

  Average scintillation indices and compute their squared *errors.*

- void **writeavg** (FILE ∗f, **scwhat** what, bool isgen, const char ∗stamp, const char ∗suffix=0, bool header=false)

  Write the average *(accumulation-time related) indices or channel fluxes on the disk.*

- double **getidx** (int meas, **scwhat** what, int i)

  Get the instant index value.

- double ∗ **getidxptr** (int meas, **scwhat** what)

  Get pointer to the array of instant scintillation indices.

- double **getavgidx** (**scwhat** what, bool isgen, int i)

  Get the averaged index value.

- double ∗ **getavgidxptr** (**scwhat** what, bool isgen)

  Get pointer to the array of average scintillation indices.

- double **geterridx** (**scwhat** what, bool isgen, int i)

  Get the relative *error of the average index.*

- double ∗ **geter2idxptr** (**scwhat** what, bool isgen)

  Get pointer to the array of squared *errors of average scintillation indices.*

- double **getmean** (bool iscor, int i, int meas=-1)

  Get the mean count in the channel in [counts per ms].

- double **getavgflux** (bool isgen, int i)

  Get the average flux in the channel in [counts per ms].

- double **geterrflux** (bool isgen, int i)

  Get the relative *error of the average flux in channel.*

- double **getsig** (int lag, int i)

  Get the second moment of counts in channel in [(counts per ms)^2].

- void **stattest** (int ndata, double microexp)

  Compute the expected scintillation indices for the results of the MASS statistical test.

- void **statflux** (int ndata, int ∗ncorr, double microexp, int k, bool isfilt=false, double ∗nonpois=0, double ∗avgflux=0, double ∗er2flux=0)

Recurrence computer of average fluxes and their squared errors *from the count series in channels.*

- int **fluxprec** (double flux, int maxprec=**sc::FLXPREC**)

  Return the decimal digits number for the FLUX representation.

Variables

- const int **DESIBIN** = 3
- const char **ERRFMT** [ ] = "%s%8.3f"
- const char **IDXFMT** [ ] = "%s%7.4f"
- const char **IDXDLM** [ ] = " "
- const char **FLXFMT** [ ] = "%s%6.*f"
- const int **FLXPREC** = 3
- const char **FLXDLM** [ ] = " "
- const char **PFMT** [ ] = "%s%5.3f"
- const char **MOMFMT** [ ] = "%s%7.0f"
- const char **MOMDLM** [ ] = " "
- const int **MODENORM** = 0
- const int **MODEGEN** = 1
- const int **MODENO** = -1
- const int **MAXLAG** = 0

5.2.1 Enumeration Type Documentation

5.2.1.1 enum sc::scwhat

Enumeration of entities to access or write on the disk, to use in get...() and write...() functions.

Enumeration values:

DSI Aperture indices writing (with DESI) or access

DESI Differential exposure indices access

AFLUX Average flux in channels access or writing

AFLUXP Average flux with non-Poisson parameter in channels writing

MOM Count moments writing

5.2.2 Function Documentation

5.2.2.1 int napcomb (int *a*) `[inline]`

Formula to get a Number of entrance apertures and their non-redundant combinations having *a* apertures

5.2.2.2 const char∗ apername (int *nchan*, int *index*)

Parameters:

nchan number of apertures

index sequential number of aperture

Returns:

pointer to name if Ok, or "" if invalid index

Name of an aperture corresponds to the given index according to convention: For index=0..nchan-1: name='A','B',... (for normal indices of scintillation in a single aperture); for index=nchan: name='AB' (first combination of apertures), nchan+1: 'AC',...,'BC','BD' and so on.

This name is copied in the static character string which contains 3 characters (two for the combination and one for trailing zero) and returns the pointer to this string. Thus, if more than one call to this function is done in one statement, the result is unpredictable.

5.2.2.3 int ind_seqnum (int *nchan*, const char ∗ *name*)

Parameters:

nchan number of channels

name ID of an aperture or the combination of apertures (e.g. "B" or "AD")

Returns:

sequential number in range [0..**napcomb**() (p. 25)-1] or -1 on error (error is set with **nr::nrerror**() (p. 83) then)

This function is needed to access certain indices with **getidx**() (p. 36), **getavgidx**() (p. 36) and **geterridx**() (p. 37) to compute their last parameter by the index "name".

See also:

apername() (p. 25)

5.2.2.4 void init (int *maxndata*, int *maxnmeas*, int *nchan*)

Parameters:

maxndata Maximal expected number of counts in base time series (one simultaneous measurement of *nchan* channels)

maxnmeas Maximal expected number of measurements (accum.time/base.time) to keep in the storage

nchan Number of channels (entrance apertures)

The module contains the global arrays which keep the normal, differential aperture and differential exposure indices computed for all apertures. These are the matrices in which the rows contain the simultaneously obtained indices for one particular base time.

For the sake of a simple code structure, some intermediate data processing is done in advance to calculation of indices itself. For this and for acceleration of logarithmic indices derivation, some work arrays are also allocated.

The dimensions of allocated arrays are determined by the numbers *maxndata*, *maxnmeas* and *nchan* which are copied in global variables which are private to the module. Another private global variable counts the performed index calculations and initialized to zero here. The number of counts in any subsequent *base-time* series should not exceed the value of the parameter *maxndata*; the same for the measurements number (base times per accumulation time) which should not exceed the parameter *maxnmeas.*

This function should be called, obviously, before the first calculation of indices.

Attention:

In addition to **init**() (p. 26) call, ALL the members of (private to module) channel parameter structures should be initialized by *nchan* calls to **chan_init**() (p. 27) made after the call to **init**() (p. 26).

If the parameters specify the storage dimensions which are less than that which were previously allocated, the storages are left unchanged.

Note:

This function must be kept in strict sync with **compute**() (p. 27).

See also:

chan_init() (p. 27) **compute**() (p. 27) **done**() (p. 31)

5.2.2.5 void chan_init (int *chan*, double *bkgr*, double *deadtime* = -1, double *nonpois* = -1, count_t * *data* = 0)

Parameters:

chan sequential number of the channel (0,1..maxnchan-1) (see **init**() (p. 26))

bkgr background level in [counts per units of *microexp* parameter in **compute**() (p. 27)]

deadtime non-linearity parameter (the dead-time of PMT) in the same units as the *microexp* parameter in **compute**() (p. 27), e.g. for [sec] it is about (20-30)*1e-9

nonpois non-poisson factor (around unity)

data pointer to the buffer - receiver of channel PMT counts

Function initializes the *chan-th* channel with the given parameters.

Attention:

This channel data are expected to be allocated by the call to **init**() (p. 26).

The assignment is made only if the respective parameter has a sense: non-negative for double-type parameters and non-null for the pointer. Thus, a single all-initializing call may be split into a sequence of calls which initialize (or update) some particular members (e.g., changing the non-linearity and background level in the channel) leaving the rest non-sense or default.

Note:

No memory allocation is done here for data[] array, it is only pointed to the buffer already allocated for receiving the channel counts.

See also:

init() (p. 26)

5.2.2.6 void compute (int *ndata*, int * *ncorr*, double *microexp*, bool *isgen*)

Parameters:

ndata Actual number of counts in the channel buffers (equal for all channels)

ncorr array of numbers of corrected counts in channels (replaced with mean if missing)

microexp Exposure time of each count (*micro-exposure)* to scale the non-linearity parameter and background in calculations (see **chan_init**() (p. 27))

isgen Generalized mode flag

This is the main utility in the module SCIND.

The parameter *ndata* must not exceed the value *maxndata* with which the **init**() (p. 26) was called; *ncorr[chan]* collects the number of counts in respective channel *chan* which are replaced with the mean value due to the data loss in a line.

This utility computes a set of all *instantaneous* (i.e. determined using individual PMT counts obtained during the *base time)* scintillation indices (SI) which can be derived for a given number of channels. Thus, both *normal* and *differential* indices (DSI) are calculated.

The implementation of this function depends on the *LOGIDX* compilation option: if it's set, then the "true" calculations of variations of logarithms of counts is implemented; otherwise - the relative differential approximation is used. Here below we refer to these alternative modes of implementation as "*if LOGIDX*".

Calculations of scintillation indices go through the following steps:

- compute the "raw" means of counts $\overline{x}$;
- if *LOGIDX*, compute logarithms of all individual counts and their means;
- calculate the dispersions and auto-covariances (lag=1,2) of counts (or - if *LOGIDX* - of their logarithms) x :

  $$\sigma_k^2(x) = \frac{1}{N-k-N_{corr}} \sum_{i=1}^{N-k} (x_i - \overline{x})(x_{i+k} - \overline{x}), \quad k = 0, 1, 2$$

  where N_{corr} is a number of pixels in the channel which were replaced with mean $\overline{x}$ due to the loss in the line (see below);
- calculate the cross-covariances of counts x and y of all combinations of channels with lags 0 and 1:

  $$\rho_0(xy) = \frac{1}{N} \sum_{i=1}^{N} (x_i - \overline{x})(y_i - \overline{y}) \ , \quad \rho_1(xy) = \frac{1}{2(N-1)} \sum_{i=1}^{N-1} (x_i - \overline{x})(y_{i+1} - \overline{y}) + (x_{i+1} - \overline{x})(y_i - \overline{y}),$$

  Note that the second formula is symmetric;
- if *LOGIDX*, *de-normalize* the dispersions and auto-covariances: multiply them by the squared mean count and by the product of mean counts in two channels, respectively ;

- correct dispersions, covariances and mean counts for non-linearity of PMT *L=deadtime/microexp* :

  $$\overline{x}_{corr} = \overline{x}(1{+}L\overline{x}) \ , \quad \sigma^2_{corr}(x) = \frac{\sigma^2(x)}{(1-2L\overline{x})^2} \ , \quad \rho_{corr}(xy) = \frac{\rho(xy)}{(1-2L\overline{x})(1-2L\overline{y})};$$

- compute scintillation indices free from photon noise (using non-Poisson factors P for conversion of means into photon noise dispersion) and influence of background *B=bkgr*microexp*, corrected to zero-exposure: normal and cross-channel (differential):

  $$s^2(x) = \frac{1.5(\sigma_0^2(x) - \overline{x}P) - 0.5\sigma_1^2}{(\overline{x} - B_x)^2}, \quad s^2(xy) = s^2(x){+}s^2(y){-}2\frac{1.5\rho_0(xy) - 0.5\rho_1(xy)}{(\overline{x} - B_x)(\overline{y} - B_y)}$$

  Here already the non-linearity corrected means, dispersions and covariances are used ;
- compute DESI for all channels, using auto-covariances with lags 0 (i.e. dispersion), 1 and 2.

  $$s^2_{de}(x) = \frac{2}{9}\left(3s^2(x) + \frac{\sigma_2^2 - 4\sigma_1^2}{(\overline{x} - B_x)^2}\right)$$

See “Multi-Aperture Scintillation Sensor. Final design report”, Appx. C (p.87).

Note on *ncorr[]*:

The counts from channels sometimes miss a certain number of data due to the data loss in the line. In this case, the ”data provider” is supposed to replace them with the average count in respective channel. Thus, *ncorr[]* is used only to correct the dispersion of counts, by decrement of the number of data in calculations.

The order of indices in which they are assigned to the elements of an output array (written to the file or exported by **getidx**() (p. 36)) is exactly the same as the order of *weights* in the weight matrix which is computed in the module WEIF: first normal indices, then the covariance of the first channel data with the second, with the third etc..., then the covariances of the second channel with third etc. See **sc::apername**() (p. 25) and **sc::ind_seqnum**() (p. 25).

Performance:

The test consists of calculation of scintillation indices for 4 channels including the DESI calculations for all channels. It takes:

- In relative approximation of indices (normalized dispersion or covariance): *1.7ms* at P-III 667 MHz,

- In true logarithmic form (*LOGIDX* definition set): *3ms* at P-III 667 MHz,

The list of features:

- the resulting indices in *scind* are **already corrected to zero-exposure**;
- The code assumes the background level and non-linearity parameter to be scaled to the micro-exposure time used for the count series. E.g. if the count (micro)exposure is 1ms, then the background level of 30 cnt/s should be set as *bkrg=30∗0.001=0.03*; deadtime of PM of 20ns - as *nonlin=20ns/0.001s=2e-5.*
- Program currently refuses to process simultaneously the count sets which have different lengths;
- The pre-allocated arrays *mean*, *cmean*, *sig* are used to store the results of statistical calculations. They contain, respectively: mean counts and means corrected for non-linearity, dispersions and covariances of counts corrected for non-linearity. Note, that, the *i-th* element of the dispersion/covariance array addresses the same channel or combination of channels as *i-th* element of the scintillation index array. The global arrays *r1* and *r2* are also used and are accessible similarly to *sig*.

The results of index calculations are put in the current measurement's row (see **getcurmeas**() (p. 30)) of the local index storing matrices.

The calculations are made irrespective of *isgen* parameter of this function. This parameter is only saved in additional local storage to be able to disentangle the normal and generalized data while averaging the indices.

Before exiting and upon the success of calculations, the counter *curmeas* of the performed measurements is incremented. If the value of *curmeas* equals or exceeds the declared in **init**() (p. 26) maximal number *maxnmeas* (the length of allocated storages for indices), then the error sc::EROFL is returned and no calculation is done.

5.2.2.7 int getcurmeas ()

Returns:
measurement number

Example: if **again**() (p. 31) was just used, the function will return "0" since no measurement were done up to now.

5.2.2.8 void again ()

Once the indices in the private module storages are not needed any more (averaged, displayed and saved to disk), the counter of current row in them should be reset to avoid the matrices overfill.

See also:
write() (p. 31) **average**() (p. 34) **init**() (p. 26)

5.2.2.9 void done ()

This function releases the memory which is allocated for index storages by **init**() (p. 26).

This function should be called only to end with calculations at all, since the modifications of arrays may be done by the calls to **init**() (p. 26) with no intermediate **done**() (p. 31).

Note:
This function must be kept in strict sync with **init**() (p. 26).

5.2.2.10 void write (FILE ∗ *f*, scwhat *what*, const char ∗ *stamp*, const char ∗ *suffix* = 0, bool *header* = false)

Parameters:

f Disk file structure pointer

what DSI: write the last obtained index set (both (D)SI and DESI), MOM: write count moments

stamp Reference string of Mode/UT to put in the first column (if *!header)* or its column name (if *header)*

suffix Last-field string of a general purpose. Will be put after the last index value or last word of the header. By default - nothing to put.

header Write header line instead of data

Function writes the last computed indices or moments on the disk in one ASCII line. The first token in the line is the reference *stamp* (which, e.g., contains the comment symbol "# ", Normal/Generalized mode symbol "N/G " and some word with the UT hour).

- if *what==sc::DSI*:

If *header* is **true**, the header is created like follows:

```
<stamp (name)> SI(A) SI(B) .. DSI(AB) ....... DESI3(A) ..
```

where SI relate to normal indices, DSI - to differential indices, DESI3 are differential exposure indices with smoothing 3-element window. Header line is kept in sync with what is written in data-writing mode (*header==false)*

For writing the index values, the format IDXFMT is used. Numbers are delimited with IDXDLM. The header-writing call is only possible after the **init**() (p. 26) was called; data-writing is allowed after at least one **compute**() (p. 27) call.

The index set is written in one line, which has a structure corresponding to that given in the header: *stamp*, *Nchannel* normal indices, *napcomb(Nchannel)-Nchannel* differential indices, and finally *Nchannel* differential exposure indices (DESI).

- if *what==sc::MOM*:

If *header* is **true**, the header is created like follows:

```
<stamp (name)> MEAN(A) MEAN(B)... SIG(A) SIG(B) .. COV(AB) ... COV1(A) ... COV2(A) ..
```

where SIG relate to dispersions and COV to covariances of channels and to auto- and cross-covariance of the signal in channels with lag 1 and 2.

For writing the moment values, the format MOMFMT is used. Numbers are delimited with MOMDLM. The header-writing call is only possible after the **init**() (p. 26) was called; data-writing is allowed after at least one **compute**() (p. 27) call.

As a last field for both *what-selections*, the *suffix* string may be printed. In MASS Software it is an optional *address* record to link the instantaneous index values with the count series data in the *count-file* from which they were derived.

- This function must be kept in strict sync with **compute**() (p. 27).
- The **write**() (p. 31) must not be called if **again**() (p. 31) was already called after the **compute**() (p. 27) call.

See also:
init() (p. 26) **compute**() (p. 27) **again**() (p. 31)

5.2.2.11 void avgmatrix (double ∗∗ *matrix*, int *ncol*, int *nrow*, signed char ∗ *rowselect*, signed char *select*, double ∗ *avg*, double ∗ *err2*, int *lag*)

Parameters:

matrix matrix[nrow][ncol] of values to average along second dimension (i.e. to obtain an average row). Meanwhile, if ncol==1, *matrix* is interpreted as a vector to be averaged into one number.

ncol number of columns in *matrix[][]*

nrow number of rows in *matrix[][]*

rowselect row selection vector[*nrow]* (set NULL to select all)

select selection value in *rowselect* (no matters if *rowselect==NULL*)

avg vector[*ncol]* of resulting average values (one value if *ncol==1)*

err2 vector[*ncol]* of resulting squared errors of avg.values (one value if *ncol==1)*

lag maximal lag in correction of error for the correlation of values

The rows are selected for averaging from the storage *matrix* by comparison of respective values in *rowselect* with *select* up in a range of the storage row numbers [0..nrow-1]. Average value is thus derived for each column for rows with index *row* complying the *rowselect[row]==select* condition.

Errors of average values are computed corrected for correlation of values as given in Korn & Korn "Spravochnik po matematike" 4th edition, section 19.8 with lag of auto-covariance up to *lag*:

$$\sigma_{\overline{x}}^2 = \frac{1}{N}\left[\sigma_x^2 + 2\sum_{k=1}^{lag}(1-\frac{k}{N})\rho_k(x)\right]$$

where the ρ_k is the covariance of values x with the lag k and σ_x^2 is the dispersion of values

$$\rho_k(x) = \frac{1}{N-k}\sum_{i=1}^{N-k}(x_i-\overline{x})(x_{i+k}-\overline{x}), \quad \sigma_x^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i-\overline{x})^2$$

If a given *lag* is more than *nrow*, the maximal lag is taken as *nrow-1.*

See also:

average (p. 34)

5.2.2.12 void average (bool *isgen*, int *lag*)

Parameters:

isgen flag that the generalized mode was used

lag Maximal counts lag to account for the index correlation in the error of mean (MAXLAG may be set)

This utility is called normally after *accumulation time* span during which rows of the index matrices are computed **getcurmeas**() (p. 30) times times. It is aimed to obtain some *mean* value of each index or, in other words, to implement the averaging of the index storage along its time-dimension. Also, the squares of uncertainties of these indices are assessed.

The computed average indices, average shifted indices, and DESIs (both normal and shifted) are put in the global vectors, to be accessed by **getavgidx**() (p. 36) and **geterridx**() (p. 37). Note, that the errors are computed and stored *squared absolute* errors, meanwhile the latter function returns the *relative* index value.

The utility **avgmatrix**() (p. 33) is used for averaging all the storages.

Note:

To the moment, the average index itself is a simple mean with no rejection; it can in principle be implemented as a mean with rejection or as a median.

5.2.2.13 void writeavg (FILE ∗ *f*, scwhat *what*, bool *isgen*, const char ∗ *stamp*, const char ∗ *suffix* = 0, bool *header* = false)

Parameters:

f Disk file structure pointer

what DSI: write the average indices, AFLUX: write the average channel fluxes, AFLUXP: write the average channel fluxes with non-Poisson values

isgen "false": write non-shifted values with errors, "true": write shifted values with errors

stamp Reference string to put in the first column

suffix Last-field string of a general purpose. Will be put after the last number in a line or after the last word of the header. By default - nothing to put.

header Write header line instead of data

Function writes the average values computed by **average**() (p. 34) and their errors on the disk in one ASCII line. The first token in the line is the reference *stamp* (line prefix and UT normally).

For writing the index values and their errors, the format IDXFMT is used; numbers are delimited with IDXDLM. For writing the fluxes and their errors, the format FLXFMT is used; numbers are delimited with FLXDLM; the non-Poisson parameter P has a fixed format "%6.4f".

The set of values is taken from shifted or non-shifted parts of storages, according to the parameter *isgen*.

The values are written in one line, which has the following structure:

- if *what==DSI*:

stamp, average normal and differential indices for all channels and their combinations, each followed by its error and average differential exposure indices (DESI) each followed by its error.

- if *what==AFLUX*:

stamp, the average fluxes each followed by its error, for all channels.

- if *what==AFLUXP*:

the same as for AFLUX, but each flux error is followed by the channel non-poissonity parameter from the channel structure (see **chan_init**() (p. 27)). It is supposed that this parameter is updated by **statflux**() (p. 41) with non-default *nonpois* parameter before.

As a last field, the *suffix* string may be printed, similarly to **write**() (p. 31). This option is added to make it possible to save the individual MASS counts in test modes, e.g. the Detector Test. The suffix may thus contain the optional *address* to link the obtained data with the count series data in the *count-file* plus the number of base-time series which results were averaged and which count buffers are written starting from this *address*.

Note:

This function must be kept in strict sync with **average**() (p. 34).

See also:

init() (p. 26)

5.2.2.14 double getidx (int *meas*, scwhat *what*, int *i*)

Parameters:

meas Measurement (row number of the index matrix)

what 0 for usual index, 1 for DESI

i index of channel or combination of channels [0..**napcomb**() (p. 25)-1]

Returns:

the index or some bad number on error

One may use **getcurmeas**() (p. 30)-1 for the parameter *meas*, to return the last obtained measurement result. Indices are ordered in the returned vector as specified in **apername**() (p. 25) documentation; the length of vector is napcomb(Nchannel) or *Nchannel* for what==1 (i.e. DESI).

5.2.2.15 double* getidxptr (int *meas*, scwhat *what*)

Parameters:

meas Measurement (row number of the index matrix)

what DSI: for usual index, DESI: for DESI

Returns:

pointer to the array of indices or to some bad number on error

This function is needed for **atm::calcint**() (p. 12)

See also:

geter2idxptr() (p. 38)

5.2.2.16 double getavgidx (scwhat *what*, bool *isgen*, int *i*)

Parameters:

what DSI: for usual index, DESI: for DESI

isgen 0 for array of non-shifted average indices; 1 for shifted indices

i index of channel or combination of channels [0..**napcomb**() (p. 25)-1]

Returns:

averaged index value or some bad number on error

The call with *isgen==true* is only possible if generalized mode was used during the last accumulation time.

See also:

getidx() (p. 36)

5.2.2.17 double* getavgidxptr (scwhat *what*, bool *isgen*)

Parameters:

what DSI: for usual index, DESI: for DESI

isgen "false" for array of non-shifted average indices; "true" for shifted indices

Returns:

pointer to the array of the average indices or to some bad number on error

This function is needed for **atm::calcn2**() (p. 13).

See also:

geter2idxptr() (p. 38)

5.2.2.18 double geterridx (scwhat *what*, bool *isgen*, int *i*)

Parameters:

what DSI: for usual index, DESI: for DESI

isgen "false" for array of non-shifted average indices; "true" for shifted indices

i index of channel or combination of channels [0..**napcomb**() (p. 25)-1]

Returns:

relative error of the average index or some bad number on error

If the stored value of the index squared error is positive, then the ratio of the square root of this squared error to the index value is returned; otherwise 0 is returned.

Note:

The squared error may, in principle, be negative due to the wrongly accounted correlation of values in **avgmatrix**() (p. 33)

See also:

getavgidx() (p. 36)

5.2.2.19 double∗ geter2idxptr (scwhat *what*, bool *isgen*)

Parameters:

what DSI: for usual index, DESI: for DESI

isgen "false" for array of non-shifted average indices; "true" for shifted indices

Returns:

pointer to the array of the errors of average indices or to some bad number on error

This function is needed for **atm::calcn2**() (p. 13).

Attention:

The internally stored array of average index errors contains the squared errors of indices computed by **avgmatrix**() (p. 33). These are the values returned here and needed for the Cn2-profile restoration. Meanwhile, the result of **geterridx**() (p. 37) is a *relative* error of index.

See also:

getavgidx() (p. 36)

5.2.2.20 double getmean (bool *iscor*, int *i*, int *meas* = -1)

Parameters:

iscor "false": return the raw mean; "true": return the non-linearity corrected mean

i index of channel [0..Nchan-1]

meas Measurement number in [0..**sc::getcurmeas**() (p. 30)-1], for iscor==true only. Last value is returned by default.

Returns:

mean count in channel during the last base time [counts per ms]

5.2.2.21 double getavgflux (bool *isgen*, int *i*)

Parameters:

isgen "false": non-shifted average counts; "true": shifted average counts

i index of channel [0..Nchan-1]

Returns:
average flux in channel [counts per ms]

Average fluxes are obtained in **average**() (p. 34).

See notes to **getidx**() (p. 36).

5.2.2.22 double geterrflux (bool *isgen*, int *i*)

Parameters:

isgen "false": non-shifted average counts; "true": shifted average counts

i index of channel [0..Nchan-1]

Returns:
relative error of average flux in channel

Average fluxes are obtained in **average**() (p. 34).

The ratio of the square root of the flux squared error to the flux value is returned.

See notes to **geterridx**() (p. 37).

5.2.2.23 double getsig (int *lag*, int *i*)

Parameters:

lag 0: simple dispersion (index<Nchannel) or covariance of counts (for returned array index more or equal to Nchannel); 1,2: the same with lag=1,2 counts

i index of channel or combination of channels [0..**napcomb**() (p. 25)-1]

Returns:
the moment value (dispersion or auto/cross-covariance)

5.2.2.24 void stattest (int *ndata*, double *microexp*)

Parameters:

ndata Number of counts in the channel buffers (equal for all channels)

microexp Exposure time of each count (*micro-exposure)* to scale the non-linearity parameter and background in calculations (see **chan_init**() (p. 27))

This is the second utility (after **compute**() (p. 27)) which deals with the channel count series.

MASS statistical test is performed using the specially modulated flux from the control LED. The current through the control LED is modulated step-like synchronously with the counting time diagram with a period of four micro-exposures. During each first and third takts of this modulation the current is at the middle level, whilst at the second and the fourth takts it is increased and decreased, respectively, relative to this middle level.

The task of this utility is to determine these middle, increased and decreased light levels and to compute the ("theoretical") expected scintillation indices (SI) and DESI which correspond to this modulation (see **compute**() (p. 27)):

$$s^2_{th} = \frac{1.5\sigma^2 - 0.5\rho_1}{\overline{flux}^2}, \qquad s^2(de)_{th} = \frac{2}{9}\left(3s^2_{th} + \frac{\rho_2 - 4\rho_1}{\overline{flux}^2}\right)$$

Here, for the specified way of modulation, the dispersion and auto-covariances with lags 1 and 2 are calculated as

$$\sigma^2 = \frac{2\Delta_0^2 + \Delta_+^2 + \Delta_-^2}{4}, \quad \rho_1 = \frac{\Delta_0(\Delta_+ + \Delta_-)}{2}, \quad \rho_2 = \frac{\Delta_0^2 + \Delta_+\Delta_-}{2},$$

and Deltas with subscripts "0", "+" and "-" are the deviations of the middle, increased and decreased light levels, respectively, from the average level. Middle level may differ from average level if the positive and the negative variations are different in magnitude.

The averaged levels are corrected for non-linearity using the channel parameters set by **chan_init**() (p. 27) and the supplied micro-exposure time *microexp*. No correction for background is made.

The call of this utility implies that the module is initialized with **init**() (p. 26) the same way as for usual measurements with **compute**() (p. 27). Since, normally, the **compute**() (p. 27) is also started on the **same** count series, the capacity *nmeas* of the index storage must be **doubled** while calling **init**() (p. 26) before the statistic test evaluation. It is assumed that **no** corrected data are present in the series (as if *ncorr* array in **compute**() (p. 27) contained zeros). So, damaged series should not be fed to **stattest**() (p. 39).

The expected values of indices are put in the index storage in the same way as **compute**() (p. 27) does. Since the flux modulation is the same in all the channels of MASS, cross-channel differential indices measured during the test (by indices()) are set to zero.

Note:

The "mode" set for the computed expected indices in the index storage is set to **sc::MODEGEN** (p. 45) to allow the simple averaging of them similarly to the case of the generalized mode measurements. Thus, the averaging of statistic test results must be done with **average**() (p. 34) with the first parameter **true** .

5.2.2.25 void statflux (int *ndata*, int * *ncorr*, double *microexp*, int *k*, bool *isfilt* = false, double * *nonpois* = 0, double * *avgflux* = 0, double * *er2flux* = 0)

Parameters:

ndata Actual number of counts in the count series stored in the channel buffers (equal for all channels)

ncorr array of numbers of corrected counts in series (i.e. replaced with mean of series if missing)

microexp Exposure time of each count (*micro-exposure)* to scale the non-linearity parameter and fluxes in calculations (see **chan_init**() (p. 27))

k sequential number of current channel series in the measurement session (i.e. this call is *k-th* to measure the statistics stored in *avg* and *er2)*

isfilt To account for the long-term modulation of the signal (clouds, etc.) in dispersion *sig*

nonpois non-Poisson parameters array[0..Nchan-1], by default - not computed

avgflux Average flux array[0..Nchan-1], by default - the local storage accessible with **getavgflux**() (p. 38)

er2flux *Squared absolute* error of flux: array[0..Nchan-1], by default - the local storage accessible with **geterrflux**() (p. 39)

This is a third utility (after **compute**() (p. 27) and **stattest**() (p. 39)) which deals with the channel data series. It is suited for calculation of the average flux in a channel and its error. Counts are assumed to be *non-correlated*, so the squared error of average is *k***ndata* times less than the count dispersion.

The results are put in the global module storage of *average* fluxes (or other place if last two or three parameters are not defaulted) or *updated* there on calls with *k>1* (see below). They are permanently available with functions **getavgflux**() (p. 38) and **geterrflux**() (p. 39) after the first call with *k=1* (unless the parameters *avgflux* and *er2flux* are specified explicitly). It is assumed that the storages for an average flux and its dispersion are allocated with **init**() (p. 26).

For any call to the function, the average count and its *dispersion* for a current series are computed like follows (hereafter *n* stays for *ndata* parameter value):

$$\overline{data} = \frac{1}{n} \sum_{i=0}^{n-1} data[i],$$

$$\sigma_{data} = \frac{1}{n-1} \sum_{i=0}^{n-1} (data[i] - \overline{data})^2$$

Here all the counts in channels are actually used corrected for non-linearity individually using the channel parameter *deadtime* initialized by **chan_init**() (p. 27):

$$data[i] = data_{raw}[i](1 + data_{raw}[i] \times deadtime/microexp)$$

Note:

The deadtime-parameter must be set to zero in **chan_init**() (p. 27) before making **statflux**() (p. 41) for determination of the non-linearity of detectors itself. It should be noticed also that the background level in channels is ignored (assumed zero). Detector non-poissonity is evidently not taken into any account but is computed and updated in channels if *nonpois* is provided as non-null.

This utility may be used either as a plain average and its error computer or as a tool for calculation of the statistics of a long series consisting of multiple data pieces (*base-time* series). In both instances, the first call to the utility must have *k=1*. If so, the average flux values and their errors in flux storages are assumed to contain no information about the previous data series. Then they are assigned from the average count and its dispersion like follows:

$$avgflux = \frac{\overline{data}}{microexp} \quad , \quad er2flux = \frac{\sigma_{data}}{n \times microexp^2} \quad ,$$

that means that we express the average flux and its error $\sqrt{er2flux}$ in [counts per unity of micro-exposure].

When one wishes to obtain the statistics of a continued series of counts (base-times of the same accumulation time, i.e. with the same source of the signal), the values in *avgflux* and *er2flux* obtained in a previous $(k-1)$-th call to **statflux**() (p. 41) are *back-converted* into the *non-normalized* counts *avgcount* and their dispersion *er2count*:

$$avgcount = avgflux \times microexp\ ,\ \ er2count = er2flux \times n(k-1) \times microexp^2$$

and updated with a current k-th series statistics like follows:

$$avgcount_{new} = \frac{1}{k}\left((k-1)avgcount + \overline{data}\right) \quad ,$$

$$er2count_{new} = \frac{1}{k}\left((k-1)er2count + \sigma_{data}\right) + Corr \quad ,$$

where the correction $Corr = (k-1) \times (avgcount_{new} - avgcount)^2$ of the dispersion for the low-frequency modulation of the average count is computed and added if *isfilt==false*. Note, that k must be incremented by user.

If the parameter *nonpois* is not null, then the estimate of the non-poissonity of the detector is computed for each channel as:

$$nonpois = \frac{er2count}{avgcount}$$

Before exiting the function, the updated parameters are again converted into the flux units similarly to the formula given above:

$$avgflux = \frac{avgcount}{microexp} \quad , \quad er2flux = \frac{sigcount}{n \times k \times microexp^2}.$$

Note:

The output results are saved in the module average flux storages **even** if the *avgflux[]* and *er2flux[]* place-holders are explicit parameters. This allows to write the **last obtained** accumulated statistics with **writeavg**() (p. 34) and access them with **getavgflux**() (p. 38), **geterrflux**() (p. 39).

Attention:

The confusion may occur on the meaning of the parameters saved in *er2flux*. This is the storage for the *squared absolute* error of average flux; meanwhile the value returned by **geterrflux**() (p. 39) is a *relative* error of flux.

5.2.2.26 int fluxprec (double *flux*, int *maxprec* = sc::FLXPREC) `[inline]`

Parameters:

flux flux value

maxprec maximal number of decimal digits

Returns:

precision to use in *printf() functions with formats like "%6.*f"

5.2.3 Variable Documentation

5.2.3.1 const int sc::DESIBIN = 3

Smoothing window width in DESI calculations. Used implicitly in **compute**() (p. 27))

5.2.3.2 const char sc::ERRFMT[] = "%s%8.3f"

Format of the relative error of any value output in the file with a leading "%s" to insert a delimiter. Length comes from the length of a longest heading

5.2.3.3 const char sc::IDXFMT[] = "%s%7.4f"

Index writing printf-format in the file with a leading "%s" to insert a delimiter. Possible "-"-sign is accounted.

5.2.3.4 const char sc::IDXDLM[] = " "

Index file delimiter

5.2.3.5 const char sc::FLXFMT[] = "%s%6.*f"

Average flux per micro-exposure writing printf-format in the file with a leading "%s" to insert a delimiter. MUST include the precision as '*'

5.2.3.6 const int sc::FLXPREC = 3

Flux accuracy: maximal number of decimal digits (see **fluxprec**() (p. 43))

5.2.3.7 const char sc::FLXDLM[] = " "

Average flux delimiter in file

5.2.3.8 const char sc::PFMT[] = "%s%5.3f"

non-Poissonity parameter writing printf-format in the file with a leading "%s" to insert a delimiter

5.2.3.9 const char sc::MOMFMT[] = "%s%7.0f"

Moments (means, covariances) writing printf-format in the file with a leading "%s" to insert a delimiter

5.2.3.10 const char sc::MOMDLM[] = " "

Moment file delimiter

5.2.3.11 const int sc::MODENORM = 0

Not-shifted flag *isgen* in **average**() (p. 34)

5.2.3.12 const int sc::MODEGEN = 1

Shifted flag *isgen* in **average**() (p. 34)

5.2.3.13 const int sc::MODENO = -1

Initial mode: unknown in isgen[]

5.2.3.14 const int sc::MAXLAG = 0

Maximal lag in calculations of the error of mean for correlated values

5.3 wf Namespace Reference

Additional to **wf_t** (p. 65), this namespace wf contains two supplementary functions - **getzshift**() (p. 47) and the sample progress indicator **progress**() (p. 48). Also, all the module constants are defined here.

Enumerations

- enum **LEFFTYPE** { **LEFFTEST** = 0, **LEFFFIX** = 1 }

Functions

- double **getzshift** (double foclen_feed, double foclen_conj)

Compute altitude shift in generalized mode from parameters of optics.

- void **progress** (int stage)

  Sample progress indicator for console.

- int **checalc** (const char ∗sedfile, const char ∗response, const char ∗wfile, int naper, double diam, double ∗eps_inn, double ∗eps_out, double z0=0, double zmax=0, double dz=0, double dzmin=0, bool calcit=false, void(∗progress)(int)=0)

  Check the weight file to correspond to given system and spectral parameters. May recalculate on mismatch.

- int **zgrid_set** (double ∗∗zgrid, double z0, double zmax, double dz, double dzmin)

  Generate the grid of values given its generation parameters.

- void **make_fft** (double ∗lambda, double ∗edist, int nedist)

  Compute the FT of SED in units of [1/cm].

- double **sfunc** (double)

  Compute the imaginary part of the Fourier transform of the light SED.

- double **sdum** (double f)

  Test-case of S-function: analytical for "quasi-gaussian" SED.

- double **flama** (double x)

  Compute the under-integral expression for the weight integration.

- double **weight** (double z, double d, double eps1, double eps2, double eps3, double eps4)

  Compute the weight of a given z *in a scintillation index.*

- void **getspechar** (double ∗lambda, double ∗edist, int nedist, double ∗leff, double ∗lblue, double ∗lred)

  Compute some characteristic points on the spectrum.

- void **read2col** (const char ∗filename, double ∗∗col1, double ∗∗col2, int ∗nrow)

  Read the two columns of argument and function values from file.

- void **conwgrid** (double ∗wgrid, int n, int scale)

  Convert array of optical (!) wavelengths into given scale.

- void **sedcrv** (double ∗slambda, double ∗sed, int nsed, double ∗rlambda, double ∗response, int nresponse)

  Multiply the spectrum curve by the response curve with optionally shifted origins.

Variables

- const double **MAX_DIAM** = 1000
- const int **MAXNZ** = 10000
- const double **WSCALE** = 1E11
- const int **WVALEN** = 15
- const double **Z0** = 0
- const double **ZMAX** = 30
- const double **DZ** = 0.2
- const double **DZMIN** = 0.04
- const double **EPSD** = 2E-2
- const double **EPSLEFF** = 0.003
- const double **EPSDZ** = 1.5
- const int **LTEXT** = 80
- const int **MKM_RANGE** = 0

5.3.1 Enumeration Type Documentation

5.3.1.1 enum wf::LEFFTYPE

Flag for **wf_t::calcleff**() (p. 74) to fix the computed spectral characteristics after comparison

Enumeration values:

LEFFTEST do not fix calculated Lblue, Lred, Leff in **wf_t** (p. 65), only compare

LEFFFIX fix wavelengths in **wf_t** (p. 65) *after* comparison

5.3.2 Function Documentation

5.3.2.1 double wf::getzshift (double *foclen_feed*, double *foclen_-conj*)

Parameters:

foclen_feed Focal length of feeding optics, [mm]

foclen_conj Focal length of conjugating lens, [mm]

Returns:

zshift Altitude shift in generalized mode, [km]

Function computes the shift from the ratio of squared focal length of feeding optics and focal length of conjugating lens. Normally, the positive lens (with a positive *foclen_conj)* is used for conjugation to negative altitudes i.e. working in a generalized mode. That's why the result is negative for positive parameters.

Note:

May depend on defocus, to be modified.

5.3.2.2 void wf::progress (int *stage*)

Parameters:

stage Part of the "work-done" (0..100) [percents]

Returns:

void

Example progress indicator, to be replaced with a suitable one

5.3.2.3 int wf::checalc (const char ∗ *sedfile*, const char ∗ *response*, const char ∗ *wfile*, int *naper*, double *diam*, double ∗ *eps_inn*, double ∗ *eps_out*, double *z0* = 0, double *zmax* = 0, double *dz* = 0, double *dzmin* = 0, bool *calcit* = false, void(∗ *progress*)(int) = 0)

Parameters:

sedfile Stellar spectrum energy distribution file

response System response curve file

wfile Weight file to check and/or put the results of calculations

naper Number of entrance apertures in scintillation device

diam see **wf_t::setaper**() (p. 68)

eps_inn see **wf_t::setaper**() (p. 68)

eps_out see **wf_t::setaper**() (p. 68)

z0 altitude grid starting point (see **wf_t::setzgrid**() (p. 70))

zmax maximal altitude in weight matrix

dz altitude step

dzmin minimal step for propostional grid (0 for equidistant)

calcit False: only check the file, True: recalculate if does not fit

progress function to call to visualize the work stage (if needed)

Returns:

the comparison result returned by **wf_t::checkfile**() (p. 72) or error code

This function makes use of all the functionalities presented by the class **wf_t** (p. 65) to compute the weight functions set for a system of a given geometry and a certain spectrum of incident light (composed by the star's SED and the system response function).

Via parameters, all the needed information on the geometry and spectrum is provided. The altitude grid, if left default (all values are 0), is taken as given by the parameters **wf::Z0** (p. 56), **wf::ZMAX** (p. 56), **wf::DZ** (p. 57), **wf::DZMIN** (p. 57).

If some mismatch is found between the weight in file *wfile* and the configuration specified by the parameters, the result "false" is returned in case *calcit=false*. The particular reason of a mismatch is provided by the call to **nr::erget**() (p. 81), **nr::ermessage**() (p. 81). Don't forget to reset then the error before continuing with **nr::erreset**() (p. 83).

If a mismatch is found and *calcit* is "true", the weights are recomputed in the file *wfile*. "True" is returned normally (the result of **wf_t::calc**() (p. 73) actually).

Note:

In case of recalculation, the "optimal" grid of altitudes specified by the parameters **wf::Z0** (p. 56), **wf::ZMAX** (p. 56), **wf::DZ** (p. 57), **wf::DZMIN** (p. 57) is taken for calculations unless the mismatch between the file and parameters was indeed the grids mismatch. In latter case, the *z0*, *zmax*, *dz* and *dzmin* parameters are taken instead.

5.3.2.4 int wf::zgrid_set (double ** *zgrid*, double *z0*, double *zmax*, double *dz*, double *dzmin*)

Parameters:

zgrid pointer double-type array

z0 starting value

zmax value to reach

dz value modifier

dzmin minimal value step

Returns:

Number of steps in z, -6 if NOT NULL zgrid pointer, -3 if bad params

This function implements the generation of a grid of abscissa values used by **wf_t::setzgrid**() (p. 70) to generate the altitude grid in weight function structure. See **weif.hxx** for description of an algorithm given in wf::setzgrid() documentation.

The function counts the steps that will be computed, allocates the double-type array (to which the first parameter must point) and fills in it. The array must be freed after usage (**free_dvector**()).

5.3.2.5 void wf::make_fft (double ∗ *lambda*, double ∗ *edist*, int *nedist*)

Parameters:

lambda wavelength grid

edist light SED

nedist w/l grid length

Returns:

0 if Ok, <0 on error

The Fast Fourrier Transform is used to compute the FT.

Before FFT, the SED is normalized by division of each value by the integral of SED. Then, the grid of SED is expanded four times with padding the obtained "tail" with zero values; SED values are divided by their argument wavelengths. Thus, the result of the FFT algorithm (**four1**() utility from NR is used) has a better sampling in the frequency domain which makes the spline interpolation of the spectrum more robust.

Since the grid of SED starts at non-zero wavelength but immediately near the wavelength of the blue-end of the sensitivity curve of MASS, the FT is made on this origin-shifted spectrum. So, the correction for this shift is needed afterwards (see below).

The units of the wavelengths are converted into [cm], the shift of the SED to its origin and the sampling of the wavelength grid are also expressed in [cm]. This ensures the correct normalization of the resulting FT and a proper correction of the real and imaginary parts of the FT for the made shift of SED to its origin. The latter correction is made in **sfunc**() (p. 51) analytically.

5.3.2.6 double wf::sfunc (double *f*)

Parameters:

f frequency in [1/cm] units

Returns:

Im(FT(SED))**2

Private function used by **flama**() (p. 52).

The results of the spline approximation of the real and imaginary parts of FT of λ_{min}-shifted SED are used. Spline coefficients are already computed in **make_fft**() (p. 50) and put into *re_edist_ftspln[]* and *im_edist_ftspln[]* arrays. Re- and Im-parts are taken from splines, corrected for the shift of SED to its origin (made while making FFT) by multiplication by $exp(-2\pi i\lambda_{min})$ and the square of the imaginary part of the product is returned.

Spline coefficients *re/im_edist_ftspln[]*, computed FT grid *re/im_edist_ft[]* and the frequency grid *edist_freq[]* are the global (private) arrays with the length *edist_ftnn*.

See also:

wf_t::calc() (p. 73)

5.3.2.7 double wf::sdum (double *f*)

Parameters:

f frequency [1/cm]

Returns:

S

Private function which may replace **sfunc**() (p. 51) in **flama**() (p. 52) for test-calculations.

In case of the SED being the "quasi-gaussian" function, the *S-function* is a purely analytical expression:

$$S = \sin^2(2\pi\lambda_0 f)exp(-4(\pi\sigma f^2)^2)/\lambda_0$$

for the SED which is a normalized λ-multiplied gaussian with a σ-dispersion centered at λ_0.

The substitution of **sdum**() (p. 51) instead of **sfunc**() (p. 51) in **flama**() (p. 52) may help to test the influence of *"non-gaussianity"* of the spectrum on the resulting shape of the weight function. The effect, in general, does not exceed a few percents (below 10 in any case) for the A0V star and the MASS response function.

5.3.2.8 double wf::flama (double *x*)

Parameters:

x spatial frequency in inverse meters

Returns:

Under-integral expression (see doc)

Private function used by **weight**() (p. 53).

In polychromatic treatment, the weight is integrated from:

$$E = f^{-8/3} * S(zf^2/2) * A(f).$$

where the function *S(f)* is a *squared* integral of SED of incident light with sine-modulation over wavelengths:

$$S(f) = \left[\int\limits_{\lambda_{min}}^{\lambda_{max}} \sin(2\pi\lambda f) F(\lambda) d\lambda \right]^2$$

which is, in fact, the squared imaginary part of the Fourier Transform of $F(\lambda)/\lambda$ computed by **sfunc**() (p. 51).

If SED is a *delta-function* centered at λ_0, then the expression turns into the known monochromatic one:

$$E = f^{-8/3} * \sin^2(zf^2\lambda_0) * A(f).$$

Here A(f) is a normalized aperture OTF for the given geometry of two annular apertures. For simplicity, the geometry parameters diameter, epsilon1..4 are transferred via the global parameters; function *S()* is also the global (private) function **sfunc**() (p. 51).

5.3.2.9 double wf::weight (double *z*, double *d*, double *eps1*, double *eps2*, double *eps3*, double *eps4*)

Parameters:

z altitude [km]

d reference size of an aperture [cm]

eps1 relative outer size of 1st (larger) aperture

eps2 relative inner size of 1st (larger) aperture

eps3 relative outer size of 2nd (smaller) aperture (may be 0)

eps4 relative inner size of 2nd (smaller) aperture (may be 0)

Returns:

Weight value in [m^(1/3)]

Private function used by **wf_t::calc**() (p. 73).

Function returns the weight for an altitude *z* in the stellar scintillation. The FT of the incident light SED divided by *lambda* is assumed to be ready and accessible by the function **sfunc**() (p. 51). The diameter of the largest entrance aperture is *d* and the radii of annular apertures are obtained by its multiplication by *eps1..4*.

The first (say, larger) aperture has the outer diameter *d*eps1* and the inner *d*eps2*. The second (smaller) aperture has the outer diameter *d*eps3* and the inner *d*eps4*.

If only two of four epsilons differ from each other (non-zero ring width) then the weighting function value for the *normal* scintillation index is returned for a given aperture (for the larger one - if eps1>eps2>0, eps3=eps4; for the smaller one - if vise versa).

If three or four epsilons are different (eps1>eps2 and eps3>eps4) then the weighting finction value for the *differential* scintillation index is returned for a pair of apertures with sizes *(d*eps1,d*eps2) – (d*eps3,d*eps4)*.

The polychromatic expression for the weight is used here, which uses the FT of the light SED to compute the so called *S-function* **sfunc**() (p. 51). This function stays for the part of the under-integral expression of the weight which is computed by **flama**() (p. 52). And the latter is already the function which is integrated by the **qromo**() (p. 80) integrator with the *relative* convergence criterion. It integrates the weight for the range of spatial frequencies from *X1=1e-4* to *X2=18*.

The weight for the zero altitude is artificially set to zero.

5.3.2.10 void wf::getspechar (double * *lambda*, double * *edist*, int *nedist*, double * *leff*, double * *lblue*, double * *lred*)

Parameters:

lambda wavelength grid

edist respective spectral energy distribution

nedist length of *lambda* and *edist* grids

leff Returned effective wavelength in [units of *lambda]*

lblue Returned wavelength of the 50% level of SED at blue side from the SED's peak in [units of *lambda]*

lred ditto at the red side from the SED's peak

Function searchs for the maximum of *edist[]* computing simultaneously the gravity center of *edist* to be returned as *leff*. Then, it goes to the left and to the right from the maximum until the level of 50% level of the maximal one is crossed by the curve. The *lblue* and *lred* points are computed by the local linear interpolation between the adjacent points.

See also:

wf_leff()

5.3.2.11 void wf::read2col (const char * *filename*, double ** *col1*, double ** *col2*, int * *nrow*)

Parameters:

filename file name

col1 pointer to the array of arguments (first column data)

col2 pointer to the array of function values (second column)

nrow pointer to the place where to put the length of allocated arrays of arguments and function values

This utility reads the pairs of numeric values from the text-type file ignoring the lines beginning with comment symbol "#". Two non-allocated pointers must be supplied as NULLs on start. Function counts the non-comment data lines, allocates the memory for arrays of argument (first column) and function values (second column) with lengths equal to the number of the counted data lines, and then reads the columns into them.

The following checks are done on each non-comment line:

- at least two numeric tokens are given in each line (argument and function)
- argument is a monotonously rising function
- argument values start from some positive number
- function is non-negative
- function has a positive average The function is thus suited for reading the spectrum-like information.

Errors possible are the problem of file opening/reading or line parsing. In case of error, the pointers are returned unallocated.

5.3.2.12 void wf::conwgrid (double ∗ *wgrid*, int *n*, int *scale*)

Parameters:

wgrid array of optical wavelengths in [angstrem] or [nm]

n its length

scale decimal logarithm of the wavelength of the infrared end of the optical range to obtain

This is an utility for conversion of the spectral curves (response function or spectrum) into the wavelength grid in a given units *scale*. It takes the value from the middle of the array and assumes that it must be in between 0.1 and 1 mkm if it's an optical range. Then it makes the conversion by multiplication of the array value by an integer power of 10 which makes the middle array value lying in the range (0.1–1.0)∗10∗∗{scale}.

Example

Let the middle point of *wgrid* (it is *wgrid[n/2])* be equal to 410 on start (i.e. the grid is in [nm]). *Scale* is given as "0", i.e. we want the spectrum to belong to the range 0.1–1.0, i.e. to be expressed in [mkm] on exit. The initial range is *range=(int)lg(410)+1=3.* So, the conversion coefficient will be *10∗∗conv*, where *conv=scale-range=-3.*

5.3.2.13 void wf::sedcrv (double ∗ *slambda*, double ∗ *sed*, int *nsed*, double ∗ *rlambda*, double ∗ *response*, int *nresponse*)

Parameters:

slambda array of wavelengths of spectrum

sed array of spectrum; returned **multiplied** by response in first *nresponse* elements

nsed length of *sed[]* and *slambda[]*

rlambda array of wavelengths of response

response array of response curve

nresponse length of *response[]* and *rlambda[]*

Function looks for the coincidence of some *k-th* element of *slambda[]* grid with the first element of the response curve wavelength grid *rlambda.* This *k* is a shift (or difference) of origins of SED and response curves (normally, the spectrum is wider).

Then, first elements of spectrum - *sed[i]*, i=0..nresponse - are rewritten with the product of SED and response curves *sed[i+k]*response[i]* which takes into account the different origins of these functions. Last *nsed-nresponse* elements of *sed[]* are replaced with zeros. Thus, on exit only first *nresponse* elements of *sed[]* make sense.

See also:
calcwf() checksed()

5.3.3 Variable Documentation

5.3.3.1 const double wf::MAX_DIAM = 1000

Maximal size of aperture [cm] (arbitrary but plausible)

5.3.3.2 const int wf::MAXNZ = 10000

Maximal number of altitude steps

5.3.3.3 const double wf::WSCALE = 1E11

Writing weights to the file : the Scale factor to divide the values by

5.3.3.4 const int wf::WVALEN = 15

Longest plausible weight value record in a file (normal format: %10.4E)

5.3.3.5 const double wf::Z0 = 0

Usual value for low boundary of altitude range

5.3.3.6 const double wf::ZMAX = 30

Maximal feasible value for upper boundary of altitude range

5.3.3.7 const double wf::DZ = 0.2

Optimal altitude modifier for z-grid creation in proportional mode

5.3.3.8 const double wf::DZMIN = 0.04

Optimal minimal altitude step for z-grid creation in proportional mode

5.3.3.9 const double wf::EPSD = 2E-2

Relative tolerance in aperture sizes to accept the weight file

5.3.3.10 const double wf::EPSLEFF = 0.003

Tolerance level in effective wavelength and SED's 50% points [mkm]. This value corresponds to change of these wavelengths which causes the change of weights not more than 1–2%.

5.3.3.11 const double wf::EPSDZ = 1.5

Relative tolerance in the altitude grid spacing: maximal acceptable ratio of *dz* and *dzmin* in file and in the comparison structure, >1

5.3.3.12 const int wf::LTEXT = 80

Length of lines in a header part of the weight file

5.3.3.13 const int wf::MKM_RANGE = 0

Units of optical range wavelength: 0 for [mkm], 3 for [nm], 4 for [\AA]

6 Part II. Class Documentation

6.1 scan_t Class Reference

`#include <scan.hxx>`

Public Types

- enum **direct**

Public Methods

- void **set** (int ncnt=0, double nonlin=-1)

  Initialize the scan arrays before accumulation.

- **scan_t** (int ncnt=0, double nonlin=-1)

  Constructor: set the private fields zero and call **set**() (p. 60).

- ~**scan_t** ()

  Destructor: deallocate the memory reserved for storages.

- void **accum** (const count_t *scan, **direct** which)

  Add the current scan to the respective scan storage.

- void **merge** ()

  Merge back- and forward scans.

- double * **get** (**direct** which=MERG) const

  Return the pointer to the needed scan.

- void **center** (double *x1, double *x2, **direct** which=MERG, bool fix-center=false)

  Locate the coordinates where the star crosses the edges of a centering triangle aperture.

- double **xshift** () const

  Return the shift of the star relative to the center of the field of view in scanning direction.

- double **yshift** () const

  Return the shift of the star relative to the center of the field of view in radial direction.

- double **defocus** (int xcenter, **direct** which=MERG, double *sharp=0) const

  Compute the sharpness of the scan and convert it into defocus.

Static Public Methods

- void **loadconst** (double scale=0, double a=0, double b=-99, double x1c=0, double x2c=0, int defrange=0, double sharp0=0, double foc-scale=0)

Assign the static calibration parameters for calculation of the star shifts and image defocussing.

- void **loadcenter** (double x1c, double x2c)

  Accept the (modified) edge-crossing coordinates for on-axis star.

- void **loadfocus** (int defrange, double sharp0=0, double focscale=0)

  Assign the new parameters for the defocussing determination.

6.1.1 Detailed Description

Usage:

For each device channel where the scanning has to be done the object of the class scan_t has to be declared. The constructor by default which is involved in declaration of the array of scan_t allocates no space. So, before accumulation of scans, the function **set**() (p. 60) is called for scan of each channel.

Each time the new scan is received, the scan accumulation **accum**() (p. 61) is called with the proper value of the flag **scan_t::direct** (p. 59) of scanning direction. After finishing the scanning, the utility **merge**() (p. 61) should be called to obtain the resulting merged scan where forward and backward scans are coadded. The utility **get**() (p. 61) returns the pointer to one of the scans (forward, backward or merged). The "scientific" functions **center**() (p. 61) and **defocus**() (p. 65) extract from the selected accumulated scan (again 3 kinds) the information on the position of the star in the centering triangle aperture and on the focus displacement, respectively. The results of **center**() (p. 61) are also stored internally and used to compute the calibrated shifts of the star **xshift**() (p. 64) and **yshift**() (p. 64). For latter two functions and for **defocus**() (p. 65), the calibration constants should be loaded with **loadconst**() (p. 62).

Before the new accumulation of scan, the function **set**() (p. 60) has to be called. If nothing has changed (dimention of scan or non-linearity), the cleaning of the storages is only done.

The scans are made of individual counts from the MASS device channels and represent the *count_t-type* arrays. The latter type definition is borrowed from the module SCIND.

6.1.2 Member Enumeration Documentation

6.1.2.1 enum scan_t::direct

which -flag to access forward scan

6.1.3 Constructor & Destructor Documentation

6.1.3.1 scan_t::scan_t (int *ncnt* = 0, double *nonlin* = -1)

The default constructor is realized with default parameters passed to **set**() (p. 60). Thus, for array of scan_t, it is obligatory to use **set**() (p. 60) after declaration; for a single scan_t instance, the constructor with proper length and non-linearity can be called immediately before **accum**() (p. 61). If default (invalid) non-linearity is supplied, it is set zero in constructor.

6.1.3.2 scan_t::∼scan_t () `[inline]`

This function releases the memory which is allocated for forward, back and merged scans by **set**() (p. 60) with zero parameter.

6.1.4 Member Function Documentation

6.1.4.1 void scan_t::set (int *ncnt* = 0, double *nonlin* = -1)

Parameters:

ncnt Number of counts in a scan (forward or backward)

nonlin non-linearity parameter for the detector (i.e. the deadtime of detector divided by the microexposure time, e.g.: *nonlin* = 24ns / 4ms = 6.0e-6)

The object contains the arrays *fscan*, *bscan* where the forward and backward scans are accumulated, and *mscan* to coadd them afterwards in a merged scan. These vectors have the logical length *ncnt*. If their physical length is smaller, they are reallocated with a new length *ncnt*.

If *ncnt* is zero, the storages are deallocated and no allocation is done (the call from destructor is done like this).

This function should be called, obviously, before the first scanning pass is finished. Default non-linearity is made invalid to leave it unchanged.

See also:

accum() (p. 61)

6.1.4.2 void scan_t::accum (const count_t * *scan*, direct *which*)

Parameters:

scan pointer to the *count_t* -type scan array

which Direction flag: FORW: forward, BACK: backward

Function adds the given array to the forward and backward scans, latter with inversed sequence of counts. Thus, forward and backward scans must be identical (see **get**() (p. 61)) if the measurement consitions are stable and no noise is taken into account. The correction for non-linearity is done with the non-linearity constant loaded in **set**() (p. 60).

6.1.4.3 void scan_t::merge ()

Function coadds the forward and backward scans into the merged scan.

See also:

accum() (p. 61)

6.1.4.4 double* scan_t::get (direct *which* = MERG) const

Parameters:

which BACK for backward scan , FORW for forward scan, MERG for merged scan

Returns:

pointer to the scan array (not to be deleted!)

6.1.4.5 void scan_t::center (double * *x1*, double * *x2*, direct *which* = MERG, bool *fixcenter* = false)

Parameters:

x1 Resulting coordinate of the crossing point of the *left* triangle edge [microstep, i.e. pixel of scan]

x2 Resulting coordinate of the crossing point of the *right* triangle edge [microstep]

which BACK for backward scan , FORW for forward scan, MERG for merged scan

fixcenter Device adjustment mode: if true, then the resulting *x1* and *x2* are treated as for the non-shifted star and saved in scan_t::x1c and scan_t::x2c. See **loadcenter**() (p. 63).

Function searches for the positions on the scan where the scan counts cross the 50%-level of the light variation range: the points where the star appears from and disappears behind the edges of the triangle aperture. "Appearance" stays for the result *x1* and has a smaller index in (forward and merged) scan than *x2* (see **xshift**() (p. 64) for the star shift calibration issue based on *x1* and *x2).*

The method is based on a linear interpolation of points on the middles of slopes of the curve. The points are taken which surround the middle of slopes. "Middle" is considered as a mean of background level (some *k-th* smallest count in a scan) and the plateau level (median of counts which have intensity more than 75% of the highest one.

Under average conditions (seeing and wandering of images = 1", scintillations of 10%, see testing main() of the module) the precision of centering is about 0.1 microsteps.

Note:

If one of the crossing points is not covered by the aperture edge while scanning, the respective coordinate is returned as zero.

Attention:

If the error is set in the system on enter (see **nrerror**() (p. 83)), the zero results are returned in *x1* and *x2*.

6.1.4.6 void scan_t::loadconst (double *scale* = 0, double *a* = 0, double *b* = -99, double *x1c* = 0, double *x2c* = 0, int *defrange* = 0, double *sharp0* = 0, double *focscale* = 0) `[static]`

Parameters:

scale Focal scale near the center [arcsec/microstep]

a Sum of tangent of angles between the edges of the triangle diaphragm and the direction to the axis of the aperture wheel

b semidifference of tangent of angles between the edges of the triangle diaphragm and the direction to the axis of the aperture wheel

x1c scan coordinate of appearance of the non-shifted star

x2c scan coordinate of disappearance of the non-shifted star

defrange Radius of sharpness calculation around the scan center [microsteps]

sharp0 Sharpness of the scan which corresponds to the perfectly focussed system

focscale focus position change per unity of the relative sharpness change (i.e. *(sh-sh0)/sh0)* taken near the focus position

The function assignes the parameters to the class variables to be used in subsequent calculations of the star displacement with **xshift**() (p. 64) and **yshift**() (p. 64). Only the non-defaulted parameters are assigned. Defaults are selected to have no sense.

Note:

These calibration parameters serve for all instances of the type scan_t because they are static. E.g. for many channels of the device, the single call to **scan_t::loadconst**() (p. 62) has to be done. For the focussing issue, the parameters (last three) have a sense for only one (say, D) aperture.

On startup, the following calibration parameters are set:

- `scale` = 1
- `A` = 2
- `B` = 0
- `x1c` = x2c = 0
- `defrange` = 17
- `sharp0` = 1.029
- `focscale` = 100.

6.1.4.7 void scan_t::loadcenter (double *x1c*, double *x2c*) `[inline, static]`

This is a shortened version of **loadconst**() (p. 62).

6.1.4.8 void scan_t::loadfocus (int *defrange*, double *sharp0* = 0, double *focscale* = 0) `[inline, static]`

This is a shortened version of **loadconst**() (p. 62).

6.1.4.9 double scan_t::xshift () const `[inline]`

The function uses the *x1* and *x2* values computed and stored in private members by **center**() (p. 61) to derive the shift of the star from the center of the field of view.

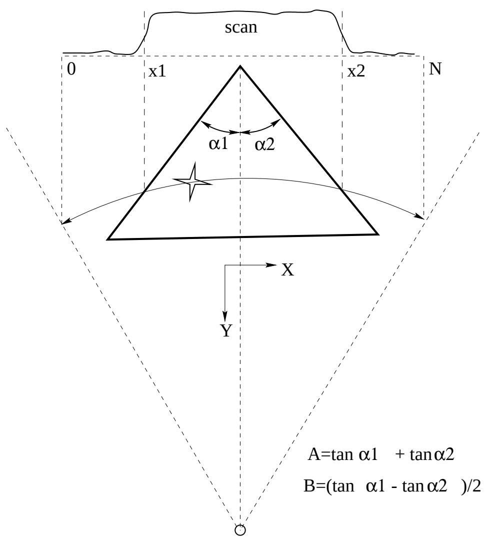

Figure 2: Scheme of MASS scanning for centering the star

As one can see from the picture, the difference of *x1* and *x2* is easily converted into the Y-shift, and the Sum of them - into the X-shift via the calibration parameters *A* and *B* (see **loadconst**() (p. 62)):

$$\Delta X = scale \cdot \frac{\Delta x1 + \Delta x2}{2} - \Delta Y \cdot B$$

Here *Dx1* and *Dx2* are the differences of the current edge-crossing coordinates *x1*, *x2* with those obtained for on-axis star.

6.1.4.10 double scan_t::yshift () const `[inline]`

The radial shift is computed as:

$$\Delta Y = scale \cdot \frac{\Delta x1 - \Delta x2}{A}$$

See also:
xshift() (p. 64)

6.1.4.11 double scan_t::defocus (int *xcenter*, direct *which* = MERG, double ∗ *sharp* = 0) const

Parameters:

xcenter scan center to compute the sharpness around

which BACK for backward scan , FORW for forward scan, MERG for merged scan

sharp resulting sharpness placeholder; not returned by default

Returns:

defocus measure in units of *focscale* in **loadconst**() (p. 62) (normally, [mm])

Function computes the *sharpness* of a scan which depends significantly on the defocussing of the system.

In current implementation, the sharpness S is the ratio of the mean square of the signal to the squared mean of the signal:

$$S = \frac{\frac{1}{2R+1}\sum_{i=Xc-R}^{Xc+R} scan[i]^2}{\overline{scan}^2}, \text{ where } \quad \overline{scan} = \frac{1}{2R+1}\sum_{i=Xc-R}^{Xc+R} scan[i]$$

I.e., the signal is taken from scan at the positions closer than *R=defrange* to the position *Xc=xcenter*.

Then, the relative sharpness deviation is computed and converted into the defocus measure:

$$Defocus = focscale \cdot \frac{S - S_0}{S_0}$$

where *So* is the focal sharpness *sharp0* (see **loadfocus**() (p. 63)).

The documentation for this class was generated from the following file:

- **scan.hxx**

6.2 wf_t Class Reference

```
#include <weif.hxx>
```

Public Methods

- **wf_t** ()

  Initialize the object fields with zero/null values.

- void **clear** ()

  Release the z-grid and weight matrix and assigns them zero.

- ∼**wf_t** ()

  Release all dynamic memory storages for deletion of object.

- void **setaper** (int nap, double diam=0, double ∗eps_in=0, double ∗eps_ou=0)

  Assign the reference size and relative diameters for apertures.

- int **getnaper** () const

  Return the number of apertures set in structure.

- int **getnw** () const

  Return the number of apertures and their combinations.

- const char ∗ **name** (int index) const

  Give a name for respective weight.

- void **setzgrid** (double z0, double zmax, double dz, double dzmin)

  Generate the altitude grid given its parameters.

- int **getnz** () const

  Return the number of altitude steps in z-grid.

- void **write** (const char ∗filename) const

  Writes weight matrix with parameters in disk ASCII file.

- int **checkfile** (const char ∗filename) const

  Checks the weight file header values to conform the wf_t *parameters (fields).*

- int **read** (const char ∗filename)

  Read the object – weight matrix and its parameters – from the disk file.

- void **calc** (double ∗lambda, double ∗edist, int nedist, void(∗progress)(int))

  Calculate the weight matrix for a given energy distribution.

- int **calcleff** (double ∗lambda, double ∗edist, int nedist, int fix)

Calculate the effective wavelength and 50%-levels of the light energy distribution and compares them with that set in member fields.

- void **interpolate** (wf_t ∗refwf, double shift, double ∗maxerr=0, int ∗iz_maxerr=0, int ∗iw_maxerr=0)

  Interpolate the weights from the reference object to the z-grid set in the object with a given optional z-shift.

- double **getleff** () const
- double **getblue** () const
- double **getred** () const
- double **getval** (int iz, int iw) const

  Get the weight value for given weight and altitude indices.

- void **copy** (int lowiz, int hiiz, int iw, double ∗dest) const

  Copy (part of) a certain weight function to double-type array.

- double **getalt** (int iz) const

  Get the altitude grid value for a given altitude index.

6.2.1 Detailed Description

This class contains all the data which describe the scintillation weighting functions of altitudes for all entrance apertures with which it is measured. The information on the apertures themselves and on the spectral characteristics of the incident light is also present in members of this class.

The allocation of the memory for double-type arrays is made in several member functions:

- **setaper**() (p. 68) - for allocation of geometry array parameters *eps_inn[], eps_out[]* ;
- **setzgrid**() (p. 70) - allocates the altitude grid *zgrid[]*;
- **calc**() (p. 73) and **read**() (p. 73) - allocate the matrix of weights before its calculation or reading, respectively.

Deallocation of all the memory allocated for arrays is done by **setaper**() (p. 68) with one zero parameter (it is called also from the destructor).

The constructor initializes the arrays and members with zeros; in all functions null pointer is a necessary demand for array to allow its allocation and, wise versa, if an array is not null, it is believed to be already allocated.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 wf_t::~wf_t () `[inline]`

The feature of **setaper**() (p. 68) to release all the memory on call with a zero argument is used.

6.2.3 Member Function Documentation

6.2.3.1 void wf_t::clear ()

The signature of allocation of memory to *zgrid* and *ws* members of wf_-t is non-zero z-dimention *nz* and non-zero respective pointers. Memory is released if only both *nz* and pointers are not zero.

6.2.3.2 void wf_t::setaper (int *nap*, double *diam* = 0, double ∗ *eps_in* = 0, double ∗ *eps_ou* = 0)

Parameters:

nap	number of annular apertures
diam	reference size or magnification factor
eps_in	array of inner diameters of apertures [cm]
eps_ou	array of outer diameters of apertures [cm]

Procedure assigns the field *naper* to the number of apertures and allocates and fills in the arrays of their relative sizes. Input arrays *eps_inn* and *eps_out* are interpreted as:

- physical instrumental sizes of apertures – then diam plays the role of the magnification of the system;
- or relative sizes (*eps_...* ≤ 1) with respect to the outer diameter of the largest aperture – then diam is a size of a largest *entrance* aperture.

In any case, only the values diam∗eps_... have a physical sense.

Usage:

- if nap<0, then the member arrays *eps_inn* and *eps_out* are simply ASSIGNED to parameters *eps_in* and *eps_ou*, respectively. In this case they are not released afterwards (see naper==0 case) – the member flag *staticapert* is set "true" for this.

- if nap>0, then input arrays are COPIED into newly *allocated* member arrays *eps_inn*, *eps_out*. *staticapert* is set "false" to allow releasing of them in destructor.

- If nap==0, then this function RELEASES all the memory allocated previously in dynamic memory for member arrays.

See also:
setzgrid() (p. 70)

6.2.3.3 int wf_t::getnaper () const `[inline]`

Returns:
Number of apertures set in wf_t

6.2.3.4 int wf_t::getnw () const `[inline]`

Returns:
Number of weights which corresponds to the apertures number set

The function which returns the number of apertures and their combinations with no difference in order is defined in **scind.hxx**: **sc::napcomb**() (p. 25).

6.2.3.5 const char* wf_t::name (int *index*) const `[inline]`

Parameters:
index sequential number of weight in weight matrix

Returns:
pointer to the static const char* name, which is empty if index is invalid

Name of the weight is a name of aperture or their combinations which corresponds to the given index. Thus, this function is just a synonym of **sc::apername**() (p. 25). In case of a problem, the error message **nr::ermessage**() (p. 81) originates from the WEIF module.

6.2.3.6 void wf_t::setzgrid (double *z0*, double *zmax*, double *dz*, double *dzmin*)

Parameters:

z0 starting altitude [km]

zmax altitude to reach [km]

dz altitude modifier ([km] for equidistant grid, unitless for proportional)

dzmin minimal altitude step [km]

Compute the grid of altitude steps for calculation of weighting functions like follows:

The grid may be either equidistant (if *dzmin* ==0) or proportional (*dzmin* !=0). The current restriction is that the altitude modifier should be positive.

Equidistant grid is given as:

$$z[i] = z0 + i * dz$$

where i is in [0,nz-1], nz conforms to the following condition:

$$z[0] + dz(nz - 2) < zmax$$

and

$$z[0] + dz(nz - 1) \geq zmax$$

Proportional grid is constructed as follows:

```
z[0] = z0
z[i] = z[i-1] + step
step0 = z[i-1] * dz
step = ( dzmin > step0 ) ? (step = dzmin) : (step = step0)
```

where i is in [0,nz-1], nz conforms to the following condition:

$$z[nz - 2] < zmax$$

and

$$z[nz - 1] \geq zmax.$$

In both types of grid, z[nz-1] is replaced with *zmax*.

This function is an interface to the lower-level private function zgrid_set() which works directly with the array *wf_t::zgrid*.

See also:

setaper() (p. 68)

6.2.3.7 int wf_t::getnz () const `[inline]`

Returns:

Number of z-steps set in object

6.2.3.8 void wf_t::write (const char * *filename*) const

Parameters:

filename file name to write weight

Procedure writes the matrix of weights which rows are preceded with the z-grid altitudes in the first column. The header is also created which contains all the parameters - the fields of wf_t.

The format of the header line is following:

```
# <KEY> <VALUE1> [<VALUE2> [VALUE3 VALUE4]]
```

where <KEY> is a single-letter designation of parameter and <VALUE> - its value(s). Keys are listed below with names of respective wf_t fields and description in brackets:

- `N` –*naper* (number of apertures)
- `A` – *D*eps_inn[0] D*eps_inn[0]* (diameters of the 1st aperture)
- `B` – *D*eps_inn[1] D*eps_inn[1]* (diameters of the 2nd aperture)
- ...
- `S` – *leff lblue lred* (effective wavelength and 50%-level points in [mkm])
- `Z` – *z0 zmax dz dzmin* (parameters for an altitude grid generation)
- `W` –*weightscale* (the scale-factor for the weight values written in this file)

After the header, the line goes which precedes the z-grid and **wf** (p. 45)-matrix values:

```
##Altit. W(A)/1E+11 W(B)/1E+11 ... W(AB)/1E+11
```

where 1E+11 is a number to which the values of weights are scaled in a file (*weightscale).*

Any similar line which starts with "##" is interpreted by **wf_t::read**() (p. 73) as a comment.

After the heading line "#Altit..." the z-grid and weight-matrix values go being separated with spaces for each altitude; thus – one line per each altitude.

6.2.3.9 int wf_t::checkfile (const char * *filename*) const

Parameters:

filename name of the file to read

Returns:

0 if parameters coincide, <0 if error or parameters do not coincide (the reason of incoincidence is reflected by following error codes: **nr::ERSPE** (p. 84), **nr::ERGEO** (p. 84) and **nr::ERZGR** (p. 84)).

Procedure reads the header info in weight file (see description of the header in **write**() (p. 71) documentation) and checks whether the parameters given in the file header coincide (within some allowed errors) with respective fields of the supplied wf_t object. The latter can be, for example, initialized from some task-file (configuration file or something else). The aim is to get a clue to decide whether the weights for given parameters can be read from a given file or should be recomputed.

The comparison is made until the first significant difference is found, in the following order: Z-grid, Apertures and Spectrum.

The major source of the difference in weights is the spectrum of light and apertures geometry. Spectral characteristics in file (3 wavelengths, see **wf_t::calcleff**() (p. 74)) must coincide to within EPSLEFF with given in *wf*; aperture diameters in wf_t and in file must differ not more than EPSD in a relative measure.

Another determinator is an altitude grid. The range boundaries in file (*z0* and *zmax)* must cover from both sides the range in wf_t to be allowed. Since the weights are normally well interpolated, the minimal spacing in altitude *dzmin* and altitude modifier *dz* in file do not have to coincide to that given in *wf*. Instead, they only have to be not more than EPSZ times larger than in *wf_t*. If the grid in the file is thinner, it is accepted.

If no error is found in header, the function **read**() (p. 73) is called on the checked file with a trial weight structure. After call this structure is cleared and resulting error (if any) is returned.

See also:

write() (p. 71) **read**() (p. 73)

6.2.3.10 int wf_t::read (const char * *filename*)

Parameters:

filename name of file to read

Returns:

0 if Ok, <0 if reading error

Procedure reads the matrix of weights and its parameters from the disk file into the fields of object. Thus, parameters are modified compared to that assigned to the structure fields before. Before reading, the procedure will try to clear all the dynamic data if the pointers of respective member arrays are not nulls.

This function is meant to be used with files which were tested by **checkfile**() (p. 72) to contain exactly the needed data. It does not check the validity of numbers in file. The weight matrix and z-grid data are converted from character representation to floating point numbers with standard function "strtod" and thus checked to be indeed numbers. Otherwise, the **nr::ERFIO** (p. 84) is returned and the altitude where the error in weight value occured.

Note:

The *zgrid[]* is read directly from the leftmost column of data section of the weight file and is NOT CHECKED to correspond to the given z-grid parameters *z0,zmax,dz,dzmin* parameters which are read from the header.

See also:

checkfile() (p. 72)

6.2.3.11 void wf_t::calc (double * *lambda*, double * *edist*, int *nedist*, void(* *progress*)(int))

Parameters:

lambda grid of wavelengths [mkm]

edist incident energy distribution (given on lambda grid)

nedist length of the *lambda* and *edist* grids

progress function to call to visualize the work stage (if needed)

Given the parameters in numeric members of the object (aperture geometry) and input spectrum, the weights are computed on a filled *zgrid* (see **setzgrid**() (p. 70)).

The weights are computed for the polychromatic case of incident light SED which involves the Fourier Transform of the SED divided by the wavelength. This FT is made by **wf::make_fft**() (p. 50) once before calculation of all weights and is represented in two global arrays as cubic spline coefficients. In order to have smoother shape, the SED is shifted to its origin (starting wavelength λ_{min}) to dump the high-frequency oscillations in FT. The back-correction for this shift is made analytically after the spline interpolation for the given frequency.

Before calculations, the weight matrix is allocated IF it was NULL. If it was not NULL, it is assumed that it was already allocated (e.g. in a previous call of this routine, say, with different en.dist.).

Then, for all apertures, their combinations and for all altitudes in the z-grid, the function **wf::weight**() (p. 53) is called to make the weight integration itself.

At the end, the SED characteristics computer **wf_t::calcleff**() (p. 74) is called to fill in the respective spectral fields. It appears that effective wavelength represents best the shape of weights if they are computed for the "quasi-gaussian" approximation of SED with its actual FWHM. Meanwhile, the deviations may be as high as 10% from the exact shape, especially for differential indices of small apertures.

The utility calls the external function *progress()* (if supplied as non-null) to visualize the degree of the made work. This degree, in percents, is an integer-type argument changing from 0 in the beginning to 100 at the end. Progress function is called twice per altitude step.

Performance

At the IBM P-III 667 MHz machine, the calculations of weights with 26 proportional steps of altitude (up to 20 km) and for 4 apertures take **14** seconds.

6.2.3.12 int wf_t::calcleff (double * *lambda*, double * *edist*, int *nedist*, int *fix*)

Parameters:

lambda grid of wavelengths [mkm]

edist incident energy distribution (SED), given on *lambda* grid

nedist number of nodes in *lambda* grid (points of en.dist.), or 0 to reset the spectral information in weight structure

fix If *fix==1*: assign the computed spectral characteristics to wf_t fields, or leave them otherwise (use **wf::LEFFFIX** (p. 47) or **wf::LEFFTEST** (p. 47))

Returns:

int(SED characteristics differ from that set in the structure) or the (negative) error code

The effective wavelength and the wavelengths at which the relative light energy is equal to 50% of the maximal are computed by the local linear interpolation for the given incident light SED. It is assumed that an array of relative intensities edist[] takes into account the instrument and atmospheric transmission functions. The computed effective wavelength and 50% points are first compared to the values in the fields *leff*, *lblue* and *lred*, and then assigned to them, if *fix==1* and (!) the difference *is* significant.

The result of comparison is returned: 0 if the difference is not significant, i.e. all three calculated numbers are within +/- *EPSLEFF* interval from the field values, and 1 if at least one of them differs by more than *EPSLEFF*. All wavelengths are in [mkm].

If the parameter \nedist is supplied as 0, the calculations and comparison are skipped and the fields *leff*, *lblue* and *lred* are reset to zero. 0 is returned.

6.2.3.13 void wf_t::interpolate (wf_t ∗ *refwf*, double *shift*, double ∗ *maxerr* = 0, int ∗ *iz_maxerr* = 0, int ∗ *iw_maxerr* = 0)

Parameters:

refwf wf_t to get the weights from

shift the shift of weight function along z-grid [km]

maxerr pointer where to put the maximal weight error due to interp (works in POLINT option of compilation; may be null).

iz_maxerr pointer to put the z-grid index of this error (may be null)

iw_maxerr pointer to put the weight index of this error (may be null)

This function uses the NR recipe for interpolation of weight values from one z-grid into another. Input weight structure wf_t must be fully determined; output structure must only contain the computed new z-grid: i.e. only the **setzgrid**() (p. 70) function must be called for it before interpolation. Other parameters must be empty - they will be assigned from the reference wf_t fields.

The parameter *shift* allows one to obtain the shifted weighting functions suited to work with the indices measured with the shifted pupil of the system (defocusing of the instrument). The *negative* altitudes when probed during the interpolation of the positively shifted functions are replaced with their absolute values. When negatively shifted function is stretched to reach the same *zmax*, the extrapolation case takes place.

The rest parameters are three pointers to variables in which the user may receive the estimated maximal interpolation error of the weight, and the indices in z-grid and weight number where this maximal error was found. For example, for 4-aperture weight matrix (10 weights for each altitude), iz_maxerr = 2 and iw_maxerr = 5 will mean that maximal error *maxerr* was found at the third stage of the NEW altitude grid *zgrid[2]* and for the weight of AC-aperture combination.

6.2.3.14 double wf_t::getleff () const `[inline]`

Return the member *leff* value

6.2.3.15 double wf_t::getblue () const `[inline]`

Return the member *lblue* value

6.2.3.16 double wf_t::getred () const `[inline]`

Return the member *lred* value

6.2.3.17 double wf_t::getval (int *iz*, int *iw*) const

Parameters:

iz altitude grid index, must be in range [0..getnz(wf)-1]

iw weight index, must be in range [0..getnw(wf)-1]

Returns:

Weight value (non-negative)

See also:

copy() (p. 76)

6.2.3.18 void wf_t::copy (int *lowiz*, int *hiiz*, int *iw*, double ∗ *dest*) const

Parameters:

lowiz low limit of the altitude index (see **getnz**() (p. 71))

hiiz upper limit of the altitude index (see **getnz**() (p. 71))

iw weight index, must be in range [0..getnw(wf)-1]

dest double-type array

This function copies one of available rows of the weight matrix into the pre-allocated destination array.

Attention:
In case of unallocated *dest* vector, the segmentation fault failure occurs! It is only checked to be non-null.

6.2.3.19 double wf_t::getalt (int *iz*) const

Parameters:
iz altitude grid index, must be in range [0..getnz(wf)-1]

Returns:
Altitude value (non-negative, in [km])

The documentation for this class was generated from the following files:

- **weif.hxx**
- **weif.cpp**

7 Part II. File Documentation

7.1 atmos.hxx File Reference

```
#include <stdio.h>
#include "nrutil.h"
#include "weif.hxx"
```

Namespaces

- namespace **atm**

7.1.1 Detailed Description

MASS project: TURBINA module file header file for atmos.cpp

The module ATMOS specifies the top-level calculations which are aimed to compute the atmospheric parameters and the turbulence profile. The weight functions related to calculations are described themselves in the module WEIF. With the help of these functions, the various integrals over the altitude which approximate the certain moments of the turbulence (calibrated into seeing, isoplanatic angles etc) are calculated and the low-resolution turbulence profile is computed.

The task of this module is split in following parts:

- using the given altitude weight functions, produce the matrix of coefficients to convert the vector of scintillation indices into a vector of the turbulence moments
- convert the vector of (base-time, instantaneous) scintillation indices into the vector of turbulence moments using the matrix multiplication
- average the set of the turbulence moments vectors and convert the average vector into a set of atmospheric integral parameters (seeing, isoplanatic angle etc.)
- fit the model with a few turbulent layers to reproduce an observed set of average scintillation indices (two methods are currently available)

Usage:

The module in current version does not need initialization, only shutdown (**atm::done**() (p. 10), see below). Note, that the initialization of the module SCIND has to be done separately while using this module in a usual pipeline of procedures within one program. Initialization and update of information on the system geometry is done only via the change of weight files provided to **atm::update**() (p. 7).

Each time the new object is selected or the altitude shift values for the generalized mode measurements are changed, the the indices-to-turbulence moments conversion matrices must be updated with **atm::update**() (p. 7) using the new weight file names and/or altitude shifts. Use **wf::checalc**() (p. 48) function of WEIF module to check the correspondence of the (file of) weight functions set to the object's spectral energy distribution (SED).

Each time some (instantaneous) scintillation indices are obtained, the computation of integrals of Cn2*h^a over atmosphere (power a=0,1,5/3,2) has to be done with **atm::calcint**() (p. 12). The results are stored internally.

After the completion of the accumulation time, the integrals of Cn2 by powers of altitude are averaged and converted into seeing, isoplanatic angle and effective altitude of turbulence by **atm::avgint**() (p. 15). Also, the procedure **atm::calcn2**() (p. 13) may be called which uses the *average* indices and their errors to restore the low-resolution Cn2-profile with one of two methods. Results are accessible with **atm::get*cn2*()** functions.

Once the profile and/or atmospheric characteristics are computed, they may be saved on disk by **atm::write**() (p. 17) with different switching keys **atm::what** (p. 7). The converted average integral values and their **relative** errors may be accessed by **atm::getval**() (p. 10) and **atm::geterr**() (p. 11) after **atm::avgint**() (p. 15) call.

Finally, to end with the measurements, call a memory destructor **atm::done**() (p. 10).

This module provides an option for a stand-alone program which accepts the weight functions file and the mass-file data (file or from standard input). On input of instantaneous scintillation indices, the turbulence moments are accumulated; on averaged indices, the profile restoration using *all* available methods happens and integrals are averaged and converted into atmospheric parameters. The needed system parameters are read from preamble-type records but may also be specified with input parameters as starting values or if no preamble-records are present. Zenith distance is not computed but taken from 'O'- and 'M'-records from the last parameter numbers (if present). See main() which should be compiled with the macro ATMOSTEST set.

Version:
1.5: Translation into C++

2.0: New decomposition, independent executable option (not just a test)

7.2 nr.h File Reference

A set of NR recipes under use in TURBINA data processing.

Defines

- #define **float** double

Functions

- float **qromo** (float(∗func)(float), float a, float b, float(∗choose)(float(∗)(float), float, float, int),double EPS)

  Modified integrator for weight calculations.

7.2.1 Detailed Description

MASS project: TURBINA module file header file for nr.c

The module NR contains the set of procedures from "Numerical Recipes in C: The art of scientific programming". All the files of the used recipes are concatenated in nr.c by means of include directives. At the top of this concatenation, in this header file, we set a heading

```
#define float double
```

to make all the mathematics done in double-precision, as accepted in MASS Software (see **nrutil.h** documentation).

The memory- and error-handling utilities are given in a separate module NRUTIL.

Usage:

The memory allocation within the MASS "scientific" modules (scind, weif, scan and atmos) is done with **dvector**(), **dmatrix**() for floating point arrays; deallocation is made with respective **free_dvector**() and **free_-dmatrix**(). Thus, we use explicitly the double-type arrays.

The only modified scientific recipe is the **qromo***()* (p. 80) integrator which is given in qromo2.c source file to implement the relative convergence criterion. The use of all the other recipes is normal, as described in the Book. They are only modified to conform C++ conventions and return after **nrerror**() (p. 83) call.

Version:

1.5: Adaptation of recipes to be compatible with C++

7.2.2 Define Documentation

7.2.2.1 #define float double

All the mathematics with "float"-type is done actually in double-type in nr.c, nrutil.cpp and MASS applications

7.2.3 Function Documentation

7.2.3.1 float qromo (float(∗ *func*)(float), float *a*, float *b*, float(∗ *choose*)(float(∗)(float), float, float, int), double *EPS*)

Parameters:

func function to integrate

a lower limit

b upper limit

choose integrator function (e.g. **midpnt**())

EPS >0: absolute convergence, as in original **qromo**() (p. 80) where EPS is a definition; <0: signals to use the relative convergence

Returns:
void

7.3 nrutil.h File Reference

memory and errors handling in NR and data processing utilities of TURBINA.

`#include <string>`

Defines

- #define **float** double

Functions

- int **erget** ()

  Get the code of error (0 if no) set in the module.

- const string & **ermessage** ()

  Get the message string set with the code returned by **erget**() (p. 81).

- const char * **ercodemessage** (int ercode)

  The error message by code.

- void **erreset** ()

  Reset the error code.

- void **nrerror** (const char *ermsg, int ercode=**recipes_ercode**)

  Error code and message setting.

Variables

- const int **recipes_ercode** = -1
- const int **ERNUL** = (-2)
- const int **ERMEM** = (-3)

- const int **ERNNL** = (-4)
- const int **ERPAR** = (-5)
- const int **ERFIO** = (-6)
- const int **EROFL** = (-7)
- const int **ERNOD** = (-8)
- const int **ERZGR** = (-9)
- const int **ERSPE** = (-10)
- const int **ERGEO** = (-11)
- const int **ERCRV** = (-12)

7.3.1 Detailed Description

MASS project: header file for nrutil.cpp and nrutil.c from Numerical Recipes, former rewritten in C++.

The module implements the utilities for handling the dynamic memory storages (vectors and matrices) in the C++ style. The "new" and "delete" instructions are used instead of "malloc" as in original NR's nrutil.c. Thus, no **nrerror**() (p. 83) is called from the utilities themselves.

In addition to reimplementation of utilities, the error handling system is made with utilities **nrerror**() (p. 83) for setting and **erget**() (p. 81), **ermessage**() (p. 81) for checking of error codes and their messages. The error codes are collected here for all the modules and made generalized in sense as much as possible. The sense of the error code may be verbalized by call to **ercodemessage**() (p. 83).

If NR_CPP macro is defined, the C-linkage conventions for NR recipes are not needed. Their source files are assumed to be converted in C++ compatible form and put in the namespace "nr", the same as one for utilities declared in this module.

Author:
N.Shatsky after NR Inc.

Version:
1.5

7.3.2 Define Documentation

7.3.2.1 #define float double

All the mathematics with "float"-type is done actually in double-type in nr.c, nrutil.cpp and MASS applications

7.3.3 Function Documentation

7.3.3.1 const char* ercodemessage (int *ercode*)

Parameters:

ercode code

Returns:

static character string which corresponds to the code

The codes are limited to the fixed set nr::ERLO..nr::ERHI; their meaning is obtained with this function. The particular cause of the error occured is rather obtained by **ermessage**() (p. 81).

7.3.3.2 void erreset ()

This function has to be called to allow the further processing of data outside the nr-module (i.e. in SCIND and ATMOS) where it is blocked on occurence of the error.

7.3.3.3 void nrerror (const char * *ermsg*, int *ercode* = recipes_-ercode)

The body of the function is evaluated if no ercode is still set, to prevent the overwriting of the information on the error which occured first. Called from NRs with no second argument. By convention, this function serves for declaring the error occured in any other "scientific" module in MASS Software - SCIND, WEIF, SCAN and ATMOS.

7.3.4 Variable Documentation

7.3.4.1 const int recipes_ercode = -1

The code set by **nrerror**() (p. 83) by default, thus - by any standard recipe.

7.3.4.2 const int ERNUL = (-2)

Error code: NULL pointer supplied

7.3.4.3 const int ERMEM = (-3)

Error code: MEMORY allocation error

7.3.4.4 const int ERNNL = (-4)

Error code: Allocation attempt on non-NULL array

7.3.4.5 const int ERPAR = (-5)

Error code: Non-sense parameter

7.3.4.6 const int ERFIO = (-6)

Error code: Bad file info or I/O error

7.3.4.7 const int EROFL = (-7)

Error code: Not expected count series (index storages overfull)

7.3.4.8 const int ERNOD = (-8)

Error code: no data supplied for calculations

7.3.4.9 const int ERZGR = (-9)

WEIF: Altitude grids are incompatible

7.3.4.10 const int ERSPE = (-10)

WEIF: Spectral Energy Distributions are incompatible

7.3.4.11 const int ERGEO = (-11)

WEIF: Apertures geometry or number are different

7.3.4.12 const int ERCRV = (-12)

ATMOS: Spectrum and response curve do not match

7.4 scan.hxx File Reference

Servo-scanning (centering, focussing) and scans reduction.

`#include <stdio.h>`

`#include "iocount.h"`

Compounds

- class **scan_t**

7.4.1 Detailed Description

MASS project: TURBINA module file header file for scan.cpp

The module SCAN contains a set of utilities which handle the scans obtained in any servo-regime of MASS device which involves the *scanning*: the counting synchronous with moving of the wheel of apertures. Here "apertures" are the focal plane diaphragms or lens settings, not the entrance apertures of the system which are played around in SCIND and WEIF modules. Servo-regimes are the focusing and centering of the MASS device apertures.

Author:
N. Shatsky, Sternberg Institute (`kolja@sai.msu.ru`)

Version:
1.5: C++ version of scan.c

7.5 scind.hxx File Reference

```
#include <stdio.h>
#include <math.h>
#include "iocount.h"
```

Namespaces

- namespace **sc**

7.5.1 Detailed Description

MASS project: TURBINA module file header file for scind.cpp

The module SCIND implements the calculation of stellar scintillation indices observed in a number of apertures (channels). The correction of observed normalized dispersion of the signal involves the *background* sky level, non-linearity of the detector and the *non-Poisson* factor (close to 1) which converts the mean of the detector signal into dispersion (the unity for an ideal detector).

As a result, the *normal* scintillation index is produced which is free from photonic statistics influences and detector imperfections. The differential indices related to the signals *covariance* of different channels are also computed. The notions of "aperture" and "channel" are identical here. The computed indices can be accessed, read or written to the disk with a number of additional utilities.

This code originates from the program **select.c** written by V.Kornilov for DASS project. All the functions and constants which deal with scintillation indices are available in the namespace "**sc** (p. 21)".

Usage:

For the sake of performance and simplicity, the work-arrays which are used for calculations of indices are allocated only once, by the initiation utility **sc::init**() (p. 26). Deallocation is devoted to done(). After init(), all the necessary computations are done by **sc::compute**() (p. 27) a certain number of times, during which the (instant) indices should be saved to disk by **sc::write**() (p. 31). If the value of some parameter of **sc::init**() (p. 26) has changed (or if even nothing has changed), this function may be called again, one does not need to call done() before. The additional space will be reallocated if needed.

The index storages filled by **sc::compute**() (p. 27) may be accessed with **sc::getidx**() (p. 36). The sequential number of a certain index in accessed array may be obtained with **sc::ind_seqnum**() (p. 25) from the name of an aperture or of the combination of two apertures. Vice versa, the character name of an index which is accessed as *i-th* in index storage is returned by **sc::apername**() (p. 25).

After some accumulation of indices (in local index storages), the indices can be averaged by **sc::average**() (p. 34). These average indices can already be used for calculations of atmosphere models and parameters (see at::calcint() and **atm::calcn2**() (p. 13) in the module ATMOS). Also, they may be saved to disk with **sc::writeavg**() (p. 34).

The counter of accumulated indices must be reset by **sc::again**() (p. 31) before beginning of the next accumulation time and then the cycle of instantaneous index computations with **sc::compute**() (p. 27) can be restarted. Alternative to **sc::again**() (p. 31) is **sc::init**() (p. 26).

After finishing the job, the memory should be cleaned with **sc::done**() (p. 31).

Note:

In principle, index averaging may be done in any moment providing thus some on-fly smoothing of data in a time-sliding window. It is possible since the index storages work as some circular buffers, i.e. the rows which are not covered in a current accumulation time still contain information from the previous accumulation time. Empty rows of index storages won't be used, since the internal storage which keeps the measurement mode is reset to "mode=unknown" each time **sc::init**() (p. 26) is called.

Attention:

Internally in the module, the counts are retained related to the original micro-exposure time, i.e. reflect the real number of pulses accumulated during the time piece. So are the counts saved in "raw moments" file (**sc::write**() (p. 31)). Meanwhile, the results of **sc::getmean**() (p. 38), **sc::getsig**() (p. 39), **sc::getavgflux**() (p. 38) are already converted into the units of micro-exposure (normally [ms] as assumed in documentation of functions in **sc** (p. 21)) which was supplied to **sc::compute**() (p. 27) and saved internally. Also, the fluxes written by **sc::writeavg**() (p. 34) are also scaled to the micro-exposure.

Author:

N. Shatsky, Sternberg Institute (`kolja@sai.msu.ru`)

Version:

1.5: Translation in C++ 1.51: a few sysnopsis changes, statflux err(flux) bug corrected

7.6 weif.cpp File Reference

```
#include <math.h>
#include "weif.hxx"
#include "nrutil.h"
#include "nr.h"
```

Namespaces

- namespace **wf**

7.6.1 Detailed Description

MASS project: TURBINA module file **weif.cpp** Description is given in **weif.hxx**

7.7 weif.hxx File Reference

```
#include "scind.hxx"
#include "nrutil.h"
```

Namespaces

- namespace **wf**

Compounds

- class **wf_t**

7.7.1 Detailed Description

MASS project: TURBINA module file header file for weif.c

This module contains the collection of utilities suited to compute the altitude weighting functions for a given set of apertures for measurement of both normal and differential scintillation indices observed with a certain bandpass*energy distribution of incident light.

Attention:
Spectral Energy Distributions must be photon numbers related, i.e. in [photons/\AA], not in [erg/\AA]. Meanwhile, they still are denoted as SED for simplicity.

This module is based on the trial program wfm2.c written by A.Tokovinin and V. Kornilov (version Feb 26, 2001). The only bug found in that program was incorrect memory freeing by the (modified) **free_vector**() function taken from Numerical Recipes (NR) package.

The module contains the definition of the weighting function class **wf_t** (p. 65) and member functions for the weight calculation and input/output. The structure bears all the data and parameters which determine the weights and the respective altitude grid on which the weight matrix is computed. The parameters determine the z-grid, the number of altitude steps, the number of annular apertures with their sizes and effective wavelength.

Usage

Prior to any operation, the constructed weight structure should be filled by **wf_t::setaper**() (p. 68), **wf_t::setzgrid**() (p. 70) and **wf_t::calcleff**() (p. 74). Being thus defined, then weight functions may be either:

- read from the chosen file : the **wf_t::checkfile**() (p. 72) is used to to check the file to coincide with the preset weight parameters (altitudes, wavelength, apertures). If the file fits the parameters, the weight function may be read from the file with by **wf_t::read**() (p. 73).
- computed by **wf_t::calc**() (p. 73).

If one gets known the new spectral energy distribution, the need to recompute the weight for this SED may be verified by **wf_t::calcleff**() (p. 74) with parameter *fix==0*. Also, the general checker/computer **wf::checalc**() (p. 48) is provided for the arbitrary input response function, SED and system geometry parameters.

The parameters - the number of apertures, resulting weights (for apertures and their combinations) and number of altitude steps - can be accessed with **wf_t::getnaper**() (p. 69), **wf_t::getnw**() (p. 69) and **wf_t::getnz**() (p. 71), respectively. The *name* of the weight is obtained by the **wf_t::name**() (p. 69) function according to the column number in the weight matrix.

Resulting weights may be shifted by altitude and/or recomputed for another altitude grid with **wf_t::interpolate**() (p. 75). Weight matrix can be written in the disk file by **wf_t::write**() (p. 71) and released with **wf_t::clear**() (p. 68). Full release of memory in weight structure is done with **wf_t::setaper**() (p. 68) with one zero parameter. The individual values of weight or altitude can be obtained with **wf_t::getval**() (p. 76) and **wf_t::getalt**() (p. 77); to copy some part of a certain weight function from the weight matrix in structure into a vector, use **wf_t::copy**() (p. 76).

The testing main() is active in module compiled as a separate executable (set WEIFTEST macro to do so). If WEIFNTEST is set zero, the executable shall be a simple weight computer given the responce and spectrum functions. If it is positive, the full circle of calculations and interpolations is done the specified number of times (say, for performance checks and memory leaks detection).

Author:

N. Shatsky, Sternberg Institute (`kolja@sai.msu.ru`)

Version:

1.5: Polychromatic precise weight, in C++

1.6: Correction of a few non-significant handling bugs. New main() for standalone weights calculations.

Index