

# **La librería de Aerotri**

**Manual para el programador**



# Contenidos

## **General . . . . . 4**

- Introducción
- Cálculo en dos fases
- Las funciones de la librería

## **La estructura TrabajoAerotri . . . . . 9**

- Introducción
- Asignar valores por defecto
- Punteros NULL
- TrabajoAerotri
- Trabajo\_Ficheros
- Trabajo\_TipoFicheros
- Trabajo\_DatosGPS
- EstimacionConjuntas
- Trabajo\_Autocalib
- Valores aproximados
- Precisiones
- Trabajo\_Opciones
- Trabajo\_Sistema
- AjbloqueConfigSalida
- Trabajo\_Salida

## **Funciones de cálculo . . . . . 28**

- InfoMalos
- FicheroBnf

## **Infobinario . . . . . 34**

- Extraer los residuos de las fotocoordenadas
- Extraer los residuos de los puntos de apoyo
- Extraer los residuos de los puntos de control
- Extraer los residuos de GPS e INS
- Residuos normalizados
- Las funciones que rellenan tablas
- Las estructuras fila

## Distorsión . . . . . 38

Lectura y escritura de ficheros  
OrientacionInterna  
InternaKPB  
Transformación entre OrientacionInterna InternaKPB  
CombinaInternas  
Modificaciones en la interna

# General

## Introducción

Para usar la librería de Aerotri ha de incluirse el fichero [aerotri\\_trabajos.h](#):

```
#include "aerotri_trabajos.h"
```

y en el momento de enlazar hacer referencia a [ATcrt.lib](#), [aerotri\\_trabajos.lib](#) y [relativa.lib](#).

La comunicación con la librería se realiza a través del llamado *trabajo de Aerotri*; en concreto, mediante la estructura en memoria de tipo **TrabajoAerotri** que representa un trabajo de Aerotri. Dicho trabajo puede almacenarse en un fichero de texto, normalmente con extensión .art.

Un trabajo de Aerotri contiene toda la información necesaria para el cálculo. Un usuario que tenga instalado Aerotri en su ordenador puede hacer doble click sobre un fichero de trabajo de Aerotri, por ejemplo *ejemplo.art*, con lo que dicho fichero se abrirá con Aerotri, e inmediatamente pulsar el botón de calcular, y se realiza el cálculo. De modo análogo, un programador que quiera emplear la librería de Aerotri para el cálculo puede simplemente 1º) llamar a la función que lee y carga en memoria un fichero de trabajo de Aerotri y 2º) llamar a la función que calcula, pasando la estructura de tipo **TrabajoAerotri** creada al leer el fichero. Se muestra un ejemplo de código que hace esto en [maintest.c](#).

El proceder anterior exige crear primero el fichero de trabajo, que incluye multitud de opciones de configuración. Más sencillo resulta pedir a la librería una estructura **TrabajoAerotri** con valores por defecto y modificar solamente lo que se quiera. Una vez hecho esto se puede llamar a la función que calcula así como a la que escribe el fichero de texto a partir de la estructura **TrabajoAerotri** en memoria. Dicho fichero podrá leerse posteriormente. El formato de los ficheros de trabajo de Aerotri no está documentado explícitamente, pero se corresponde casi exactamente punto por punto con la estructura **TrabajoAerotri** en memoria, que se documenta aquí. Además, al ser un fichero de texto es muy fácil

de interpretar. Por último, es de mucha ayuda modificar parámetros en la interfaz gráfica de Aerotri y analizar las variaciones en el fichero .art correspondiente. Aerotri genera ese fichero cada vez que se pulsa el botón «Calcular» aunque no exista licencia de Aerotri.

Así pues, la manera más sencilla de emplear la librería sería como sigue :

```
int nret;
char16_t mensaje[512];
TrabajoAerotri *tr=get_trabajo_aerotri();
tr->ficheros.fotos=u"estatua.ftm";
tr->ficheros.apoyo=u"puntos_apoyo.xyz";
nret=calcula_trabajo_aerotri(NULL,0,tr,mensaje,ATIDIOMA_Esp);
ATtrabajos_free(tr); //This is the same as free(tr), except that the code is
                     //executed in the aerotri_trabajos.dll module
```

El programador puede crear él mismo una estructura **TrabajoAerotri** sin más que declarar un objeto de ese tipo :

```
TrabajoAerotri trabajo;
```

rellenarlo y pasar punteros a ese objeto a las funciones de la librería. Véase la sección *Asignar valores por defecto* del siguiente capítulo.

Esta estructura, a su vez, contiene punteros a memoria que se reserva; por ejemplo, para los nombres de los ficheros de entrada. Si es el programador el que maneja esos punteros NO debe llamar a la función *libera\_trabajoAT*, mientras que si es la librería la que ha reservado memoria para los mismos, lo que solamente sucede si la estructura se rellena mediante una llamada a *lee\_trabajoAT*, ha de llamar a dicha función para liberarlos :

```
TrabajoAerotri trabajo;
/* Here the programmer must set all pointers within trabajo to NULL, viz :
    trabajo.ficheros.fotos=NULL;
    [...]
    trabajo.configsalida.fich_Imaster.rutaf=NULL;
    The next chapter lists all those pointers.
*/
lee_trabajoAT(L"ejemplo.art",&trabajo);
/*
...
*/
libera_trabajoAT(&trabajo);
```

Si el programador rellena él mismo la estructura **TrabajoAerotri** el modo de trabajar será algo así como

```

int nret;
char16_t mensaje[512];
TrabajoAerotri trabajo;

RellenaMiTrabajo(&trabajo); //rellena la estructura con la información
                             //para el cálculo
nret=calcula_trabajo_aerotri(NULL,0,tr,mensaje,{ATIDIOMA_Esp});
if(nret=={AT_ESC}){ //The user escaped
    /*...*/
}else{
    /*... analizar los otros valores de nret
    if(...)
    }else if(...)
    }...
    */
}
LiberaMiTrabajo(&trabajo);

```

Las funciones *RellenaMiTrabajo* y *LiberaMiTrabajo* son nombres inventados en cuyo lugar ha de ir la función o funciones que el usuario de la librería haya programado.

## Cálculo en dos fases

En lugar de *calcula\_trabajo\_aerotri* tal vez prefiera un cálculo separado de los valores aproximados y el ajuste del bloque:

```

int nret;
struct strRelativaParams *rparams;
struct strAbsolutaCallData *aparams;
TrabajoAerotri trabajo;
/* ... */

rparams=get_relativa_params();
aparams=get_absoluta_calldata();
if(rparams==NULL{} || aparams==NULL){
    ATtrabajos_free(aparams); //frees if not NULL
    ATtrabajos_free(rparams); //ditto
    /* El programa se ha quedado sin memoria. Mostrar mensaje de error */
}

RellenaMiTrabajo(&trabajo);
nret=prepara_trabajo_relativa(&trabajo, &rparams);

```

```

if(nret!=0){
    /*Mostrar mensaje de error según nret*/
}
nret=calcula_relativa(/* ... */);
if(nret!=0){
    /*Mostrar mensaje de error según nret*/
}
trabajo.ficheros.aproximadas=/* ... */ //indicar el fichero de valores aproxima-
//dos generado en el cálculo anterior; es decir, en calcula_relativa().

nret=prepara_trabajo_absoluta(&trabajo, &aparams);
if(nret!=0){
    /*Mostrar mensaje de error según nret*/
}
nret=calcula_absoluta(/* ... */);
if(nret=={AT_ESC}){
    /*No hacer nada*/
}else if(nret!=0){
    /*Mostrar mensaje de error según nret*/
}else{
    /*Mostrar mensaje de que el cálculo ha terminado
    bien y realizar otras tareas*/
}

ATtrabajos_free(aparams);
ATtrabajos_free(rparams);

```

## Las funciones de la librería

Con lo explicado en las secciones anteriores y los comentarios en [aerotri\\_trabajos.h](#) es casi suficiente. Solamente falta aclarar el comportamiento de las funciones *lee\_trabajoAT* y *change\_absparams* y las estructuras devueltas por *calcula\_relativa* y *calcula\_absoluta* en punteros pasados como parámetros, que se explican más adelante en capítulo a parte.

### lee\_trabajoAT

Lo primero que hace esta función es liberar los punteros de la estructura en caso de que no sean **NULL**. De esta manera, si se deja que sea la librería la que cree la estructura, mediante *get\_trabajo\_aerotri*, se rellena y modifica solamente mediante *lee\_trabajoAT*, y por último se libera su contenido mediante *libera\_trabajoAT* y la propia estructura mediante *ATtrabajos\_free*, el programador no tiene que preocuparse en ningún momento de dichos punteros. En cambio, si es el programador el asigna valores a esos punteros, o bien crea

un objeto de ese tipo, antes de cualquier llamada a *lee\_trabajoAT* ha de poner a **NULL** todos los punteros :

```
TrabajoAerotri trabajo;  
int err_code;  
  
setNULL(&trabajo); /* Poner a NULL los punteros que contiene la estructura */  
err_code=lee_trabajoAT("ejemplo.art",&trabajo);
```

*setNULL* es un nombre inventado. Se trataría de una función del programador que pone los punteros a **NULL**. En el siguiente capítulo se enumeran todos los punteros a memoria que tiene la estructura.

La función *lee\_trabajoAT* efectúa las siguientes acciones en el orden indicado :

- 1º. Liberar todos los punteros de la estructura que no sean **NULL**
- 2º. Rellenar toda la estructura con valores por defecto
- 3º. Abrir y leer el fichero

Este orden de acciones permite emplear esta función para asignar valores por defecto a una estructura **TrabajoAerotri** sin más que llamarla pasando dicha estructura y un nombre de fichero inexistente.

## change\_absparams

Esta función permite modificar una estructura de tipo **struct strAbsolutaCallData**. Antes de continuar leyendo tenga en cuenta que es posible que nunca necesite lo que aquí se explica.

De momento esta función solamente puede modificar un campo de la estructura **strAbsolutaCallData**. La razón de proporcionar esta función para modificar ese campo en particular es que dicho campo siempre adquiere el valor cero cuando la estructura se crea a partir de un **TrabajoAerotri** mediante la llamada a *prepara\_trabajo\_absoluta*, que es la única manera de rellenar la estructura (la librería de Aerotri la declara pero no proporciona su definición). Por tanto la única manera de poner el susodicho campo a uno es mediante esta función. Así que las dos únicas maneras de llamar a la función y que esta haga algo son

```
change_absparams(params,{APARAMS_flags0_Semilibre},1);  
change_absparams(params,{APARAMS_flags0_Semilibre},0);
```

en donde *params* es de tipo `struct strAbsolutaCallData*`.

Establecer la bandera **APARAMS\_flags0\_Semilibre** a uno indica a la función de ajuste del bloque que continúe con el ajuste aunque los puntos de control y de GPS, cuando se indica alguno de estos dos ficheros o ambos, no sean suficientes para llevar el bloque a su



sistema de coordenadas 3D (es decir, que no haya puntos comunes suficientes entre estos ficheros y el bloque que se ajusta). En caso de que no se incluya ni uno ni otro fichero no es necesario establecer esta bandera; el ajuste continúa sin más en un sistema arbitrario. Si la bandera está a cero, se indica un fichero de apoyo o GPS, y los puntos en esos ficheros son insuficientes, la función *calcula\_trabajo\_absoluta* devolverá 12 según se explica en [clases\\_mainabsoluta.h](#).

Cuando la bandera no está activa (es el comportamiento por defecto) y la función devuelve el valor 12, normalmente será señal de un error en el fichero de puntos de apoyo o GPS, como que el fichero indicado es el correspondiente a otro trabajo o que los puntos o centros de toma no se llaman igual en estos ficheros y en el fichero de fotogramas, como por ejemplo 1034 frente a 001034, además de que cuando se indican esos ficheros obviamente se quiere que el bloque quede calculado en el sistema de coordenadas de los mismos. Por lo tanto un valor devuelto de 12 realmente refleja un error, por lo que seguramente nunca deba preocuparse por esta bandera.

## La estructura TrabajoAerotri

### Introducción

La estructura **TrabajoAerotri** representa completamente un trabajo de Aerotri y es todo lo que se necesita para interactuar con las funciones de la librería Aerotri-lib. El fichero [ClaseTrabajo.h](#) y los ficheros por este incluidos tienen algunos comentarios aclaratorios, pero que no bastan para entender todos los campos de la estructura. Este capítulo se dedica a su descripción completa.

Un trabajo de Aerotri guarda más información de la necesaria para un cálculo. La razón es que tiene que almacenar completamente el estado de todos los controles, incluidos los desactivados y aquellos que no van a intervenir en el ajuste. Por ejemplo, si no se indica fichero de GPS/INS todas las opciones relativas al GPS se pueden ignorar para el cálculo, pero el trabajo de Aerotri las guardará igual. Otro ejemplo: si se selecciona como sistema de coordenadas *Conforme Genérico* todas las opciones del sistema UTM no intervienen, pero el **TrabajoAerotri** ha de almacenarlas igual, de modo que si el usuario selecciona UTM los parámetros de esta proyección no cambien desde la última vez que lo seleccionó, etc. Si se usa la estructura para pasarla a las funciones de la librería que calculan no es necesario asignar los valores que serán ignorados.

## Asignar valores por defecto

Puesto que la estructura y sus sub-estructuras contienen en total muchos campos puede ser pesado asignar valores por defecto a todos ellos, así como difícil de hacer correctamente para el que lo aborde por primera vez. La librería de Aerotri no proporciona una función cuya finalidad sea explícitamente iniciar una estructura con valores por defecto, pero se puede conseguir el mismo objetivo llamando a *lee\_trabajoAT* indicando un nombre de fichero que no exista, o simplemente mediante una llamada a *get\_trabajo\_aerotri*:

```
TrabajoAerotri trabajo;  
/*Set all pointers to NULL. Code for that omitted*/  
lee_trabajoAT(L"", &trabajo); //Ignore the returned value  
  
/*O bien*/  
TrabajoAerotri *trabajo;  
trabajo=get_trabajo_aerotri(); //Rellena con los valores por defecto
```

La primera manera requiere que el programador ponga a **NULL** todos los punteros, ya que lo primero que hace la función *lee\_trabajoAT* es liberar dichos punteros si no son **NULL**. En la segunda manera es la propia función *get\_trabajo\_aerotri* la que inicializa los punteros.

## Punteros NULL

Se ha hecho mención varias veces a los punteros de la estructura, que si no son **NULL** son liberados por las funciones *lee\_trabajoAT* y *libera\_trabajoAT* (la primera de estas funciones llama a la segunda para liberar la estructura que se le pasa). Pues bien, dichos punteros son los que se muestran a continuación, mediante un ejemplo que los pone todos a **NULL**:

```
TrabajoAerotri trabajo;  
  
trabajo.ficheros.fotos=NULL;  
trabajo.ficheros.aproximadas=NULL;  
trabajo.ficheros.apoyo=NULL;  
trabajo.ficheros.gpsins=NULL;  
trabajo.ficheros.interna=NULL;  
trabajo.configsalida.fich_Digi.rutarel=NULL;  
trabajo.configsalida.fich_Digi.rutaf=NULL;  
trabajo.configsalida.fich_Imaster.rutarel=NULL;  
trabajo.configsalida.fich_Imaster.rutaf=NULL;
```

El programador atento habrá observado que si inicializa la estructura llamando a la función *lee\_trabajoAT* pasando un nombre de fichero inexistente, los punteros de *fich\_Digi* y *fich\_Imaster* no se establecen a **NULL** sino que apuntan a rutas por defecto, como por

ejemplo ".\\". Estas cadenas son posiciones de memoria estáticas dentro de la librería de Aerotri, por lo que se trata en cualquier caso de memoria que no se ha reservado y que la función *libera\_trabajoAT* no va a liberar. El programador no tiene que preocuparse en absoluto de esto.

## TrabajoAerotri

```
typedef struct{
    u8int sensor;
    u8int modo_toma;
    u8int modo_medida;
    Trabajo_Ficheros ficheros;
    Trabajo_TipoFicheros tiposf;
    Trabajo_DatosGPS datosgps;
    EstimacionConjuntas conjuntas;
    Trabajo_AutoCalib autocalib;
    ValoresAproximados valoresaproximados;
    Precisiones apriori;
    Trabajo_Opciones opciones;
    Trabajo_Sistema sistema;
    AjbloqueConfigSalida configsalida;
    Trabajo_Salida salidatrabajo;
} TrabajoAerotri;

#define ATMODOTOMA_Unknown    0
#define ATMODOTOMA_Plano      1
#define ATMODOTOMA_Generico    2
#define ATMODOMEDIDA_Unknown  0
#define ATMODOMEDIDA_Manual    1
#define ATMODOMEDIDA_Automatico 2 //Puede incluir puntos manuales también

sensor          0, cónico; 1, modelo local (usar siempre 0)
modo_toma        Indica el modo en que fueron tomadas las fotografías. Se emplea
                  solamente en el cálculo de valores aproximados y no tiene mucha
                  influencia.
modo_medida      Aún no se emplea, pero versiones futuras de la librería segura-
                  mente lo empleen para calcular de una u otra manera según
                  la medida sea manual o automática, especialmente en el ajuste
                  robusto.
```

La variable *modo\_toma* se emplea en el cálculo de valores aproximados. El modo *Unknown* equivale esencialmente al modo *Generico*. En el caso de que las fotografías estén tomadas en paralelo y aproximadamente en el mismo plano, como habitualmente sucede

en fotogrametría aérea, el indicar `ATMODOTOMA_Plano` favorece que no se produzcan errores en el cálculo de valores aproximados.

## Trabajo\_Ficheros

```
typedef struct{
    char16_t *fotos, //Puntero al fichero de fotogramas
              *aproximadas, //Puntero al fichero de valores aproximados
              *apoyo, //Puntero al fichero de apoyo
              *gpsins, //Puntero al fichero de gps/ins
              *interna; //Puntero al fichero de orientación interna (fichero de cámara)
    uint8_t generar;
} Trabajo_Ficheros;
```

Esta estructura contiene punteros a los nombres de los ficheros de entrada más la variable `generar`. Algunos ficheros pueden no existir. En ese caso los punteros correspondientes deben estar a `NULL`.

El fichero `fotos` es obligatorio siempre. `aproximadas` es obligatorio para el ajuste del bloque. Si no existe, la función `calcula_trabajo_aerotri` lo calculará automáticamente, por lo que se puede pasar `NULL` en ese parámetro a esa función. La presencia de los parámetros obligatorios se comprueba en las funciones `prepara_trabajo_relativa`, `prepara_trabajo_absoluta` y `calcula_trabajo_aerotri`.

La reducción al punto principal y corrección de la distorsión indicadas en el fichero `interna` se aplicarán a las coordenadas del fichero de fotocoordenadas. Por tanto el fichero `interna` no se ha de indicar si las coordenadas del fichero de fotogramas ya están refinadas, esto es, respecto al punto principal y con la distorsión corregida. Se recomienda que las fotocoordenadas se indiquen en coordenadas píxel brutas, respecto a la esquina superior izquierda de la foto, y se indique en `interna` la orientación interna de la cámara. De esta manera, si en el ajuste de la aerotriangulación se calculan parámetros de autocalibración, Aerotri podrá combinar el fichero de `interna` indicado con los parámetros calculados para generar un fichero de `interna` nuevo.

Si el fichero de fotogramas es de formato Aerotri o PATB para cada fotograma se indica un valor de la focal. Si se especifica un fichero de `interna` el valor de la focal en este tiene preferencia sobre los valores indicados en el fichero de fotogramas.

Si no se indica ni fichero de puntos de apoyo ni fichero de `gps/ins` el cálculo se hace en un sistema arbitrario. Si se indica al menos uno de estos ficheros pero no hay puntos suficientes para el cálculo el cálculo no se realiza y se devuelve un código de error. La razón es que cuando esto sucede normalmente es porque los ficheros indicados no son los correctos. Para cambiar este comportamiento y que la librería continúe con el cálculo a

pesar de esta circunstancia véase cómo hacerlo en la sección *change\_abbrevparams* del capítulo anterior.

generar es una variable cuyos bytes inferiores son flags, que si están a 1 indican si se ha de generar un fichero con formato de Aerotri de un cierto tipo (por ejemplo, de gps/ins) en caso de que el correspondiente fichero de entrada no sea de Aerotri. El programa Aerotri pasa 5 en esta variable, que es 00101 binario, lo que significa que se generen ficheros de Aerotri de fotogramas y apoyo. La mayoría de los formatos de ficheros de fotogramas no indican ningún valor para la focal de las fotografías, ya que entienden que esa información estará en un fichero de interna. En estos casos en el fichero de formato Aerotri generado el valor de la focal para todos los fotogramas será de 1.0. Según lo indicado dos párrafos más arriba esto no importa porque el valor de la focal se tomará del fichero de interna.

## Trabajo\_TipoFicheros

```
typedef struct{
    uint8_t tfot,tapr,tapy,tgps,tint,
            tdapr;
} Trabajo_TipoFicheros;
```

Esta estructura indica el formato de cada fichero de entrada, más la variable tdapr. Los posibles valores para cada una de las variables son

tfot	(0,)1 : Aerotri, 2 : PatB, 3 : Image Master, 4 : Enso Mosaic
tapr	(0,)1(,2) : Aerotri
tapy	(0,)1 : Aerotri, 2 : XYZ
tgps	2 : Aerotri, 4 : AeroOffice, 5 : TopoSys EO-file
tint	0 : Aerotri, 1 : ini, 2 : Image Master, 3 : RapidCal

Si un fichero de entrada no existe su correspondiente variable en esta estructura será ignorado. Los valores para los formatos Aerotri son resultado de la historia del programa y la compatibilidad de versiones nuevas con ficheros de trabajo antiguos. Los valores entre paréntesis no se han de emplear y no los generará nunca la GUI de Aerotri. Los distintos formatos de cada clase de ficheros se encuentran descritos en el manual de Aerotri.

tdapr	Tipo de valores aproximados. 0 : Unos cualesquiera; 2 : Los valores ajustados de un ajuste anterior
-------	-----------------------------------------------------------------------------------------------------

El valor 1 para tdapr lo emplea internamente Aerotri. Si se pasa 2 en tdapr pero Aerotri ve que algunas opciones del ajuste no coinciden con las del ajuste anterior según el fichero de valores aproximados (que si tdapr es 2 tiene que ser el de valores ajustados del ajuste anterior), como por ejemplo el sistema de coordenadas o los parámetros de autocalibración, lo «degrada» a 1.

## Trabajo\_DatosGPS

```
typedef struct{
```

```
    u8int  tdgps,tdins;
```

```
    bint  signoins;
```

```
} Trabajo_DatosGPS;
```

```
#define ATTrabajo_TDGPS_Offset 255      Error constante global
```

```
#define ATTrabajo_TDGPS_NoError 0
```

```
#define ATTrabajo_TDGPS_Constant 1
```

```
#define ATTrabajo_TDGPS_Linear 2
```

tdgps,tdins      Tipo de errores sistemáticos presentes en los datos gps e ins. 0, sin error; 1, error constante; 2, error linear; 255, offset

signoins          0: el criterio interno de Aerotri; 1: el criterio más habitual en los ficheros de ins

Las componentes de error sistemático son independientes para cada pasada; es decir, se calcula un juego de parámetros para cada pasada. El tipo de error *offset* significa un error constante e igual para todas las pasadas, y solamente se aplica a los datos INS. En principio el tipo de errores esperable es 1/0, es decir, constante para el GPS y sin error para el INS, o bien 2/0 si el cálculo de los datos GPS no tiene mucha calidad. Para el INS el error puede ser *offset* si no está bien calibrada la posición relativa (alineación) del sistema inercial con los ejes de la cámara, y *constante* si el programa que ha generado el fichero INS no ha transformado bien los giros. En caso de no querer preguntar al usuario lo más seguro es pasar 2/1, si bien 1/*offset* será prácticamente siempre suficiente. Aerotri por defecto asigna valores 1/1.

El valor de *ATTrabajo\_TDGPS\_Offset* es arbitrario. Para los demás se puede emplear las macros o bien los números 0, 1, 2, ya que no van a cambiar pues tienen un significado que permite emplearlos en operaciones aritmética, y el código interno de Aerotri lo hace. El valor es igual al número de grados de libertad del error sistemático para cada coordenada.

Para *signoins* se ha de pasar 1. Un 0 indica un sentido de giros positivos que no es el empleado en fotogrametría, si bien es más habitual en matemáticas y es el que emplea internamente Aerotri.

## EstimacionConjuntas

```
typedef struct{
```

```
    u8int  xy;
```

```
    u8int  XY;
```

```
    u8int  planiZ;
```

```
} EstimacionConjuntas;
```

Estos parámetros tienen significado en un ajuste robusto y se ignoran si el ajuste es mínimo cuadrático. Indican si un error muy grande en una coordenada de un punto significa que también se ha de eliminar la otra coordenada; es decir, indica un tratamiento conjunto o individual de las coordenadas de los puntos de cara a la estimación robusta.

$xy$  se refiere a las fotocoordenadas  $x$  e  $y$ ,  $XY$  a las coordenadas planimétricas de los puntos de apoyo y  $planiZ$  a las coordenadas planimétricas por un lado frente a la coordenada  $Z$  por el otro. Lo habitual es que los valores correctos sean 1/1/0; es decir, que los errores en las fotocoordenadas  $x$  e  $y$  no son independientes; tampoco los errores en  $X$  e  $Y$  de los puntos de apoyo, pero el error en  $Z$  de los puntos de apoyo sí es independiente del error en planimetría.

## Trabajo\_Autocalib

```
typedef struct {
    bint existe;          //Si se ha seleccionado Autocalibración para el cálculo
    struct {
        bint f;
        bint pp;
        bint distorsion;
        struct {
            bint radsim;
            bint tansim;
            bint asimetricas;
        } componentes;
    } selected;
    Config_distorsion config_dist;
} Trabajo_AutoCalib;
```

Si la variable `existe` está a *false* el resto de la estructura ya no se lee en las funciones que calculan, pero obviamente sí en la función `escribe_trabajoAT`. Las variables dentro de la estructura `selected` indican si unos u otros parámetros se han seleccionado o no para el cálculo:

```
bint f;          Longitud focal
bint pp;         Punto principal
bint distorsion; Parámetros de distorsión
```

Si `distorsion` está a *false* no se incluyen componentes de distorsión en el cálculo. Además de la distorsión la autocalibración puede incluir el cálculo de la longitud focal y el punto principal. Si `distorsion` está a *true*, la estructura `componentes` indica para cada uno de los tres juegos de componentes si ha sido seleccionado:

```
radsim          Distorsión radial simétrica
```

tansim	Distorsión tangencial simétrica
asimetricas	Distorsiones asimétricas

El detalle de cada componente se incluye en config\_dist. Por ejemplo, ciertos bits dentro de config\_dist indicarán si cada una de las componentes  $a_2... a_5$  de la distorsión radial simétrica se ha seleccionado para su cálculo, para lo cual es necesario además que radsim sea *true*.

## Config\_distorsion

### typedef struct{

```
float semidiag; //Semidiagonal del fotograma, e.d.,  $r_{\text{máx}}$ ,
uint8_t modelo_poli; //modelo polínomico.
uint8_t modelo_asim; //modelo de distorsiones asimétricas.
uint8_t condicion_radsim; //condición para la definición del término lineal de la Drs
float r1;
uint16_t param_radsim; //flags que indican los parámetros que se calculan de la Drs.
     $a_1, a_2, a_3...$  empezando en el bit más bajo.
uint16_t param_tansim; //Idem para la distorsión tangencial simétrica.
uint param_asim1; //Idem para la primera serie de componentes asimétricas.
uint param_asim2; //Idem para la segunda serie de componentes asimétricas.
```

### } Config\_distorsion;

```
#define CAL_MODPOL_IMPAR 1
#define CAL_MODPOL_COMPLETO 2
#define CAL_MODASIM_RADTAN 1 //radial/tangencial
#define CAL_MODASIM_VECTOR 2 //vector giratorio
#define CAL_CONDICIONR_ORTO 0 //que la distorsión radial sea ortogonal a la focal
    ( $a_1 = 0$ )
#define CAL_CONDICIONR_R1 1 //ajustar  $a_1$  para que Drs( $r_1$ )=0
#define CAL_CONDICIONR_MM 2 //ajustar  $a_1$  para que |Drs mín| = |Drs máx|
```

semidiag	Por lo general se deja que el programa deduzca el valor adecuado de $r_{\text{máx}}$ . Buscará el valor máximo de $r$ y tomará un valor redondo que se ajuste a ese valor. Para ello se ha de pasar infinito o NaN, lo cual se puede hacer con las macros de <code>system.h</code> . En concreto, el código <code>NAN_F(semidiag)</code> asigna a semidiag un NaN.
modelo_poli	Por compatibilidad con otros modelos de distorsión se recomienda emplear siempre <code>CAL_MODPOL_IMPAR</code> .
modelo_asim	Salvo que haya alguna razón para lo contrario se ha de pasar <code>CAL_MODASIM_RADTAN</code> .
condicion_radsim	Se recomienda emplear siempre <code>CAL_CONDICIONR_ORTO</code> .



<code>r1</code>	Si <code>condicion_radsim</code> es <code>CAL_CONDICIONR_R1</code> este es el valor para el que se quiere $Drs(r1)=0$ . En particular, si <code>condicion_radsim</code> es <code>CAL_CONDICIONR_ORTO</code> este valor es ignorado.
<code>param_radsim</code>	En caso de que se calculen parámetros de distorsión radial simétrica este parámetro indican qué componentes se calculan. Si el bit más bajo está a 1 indica que se ha de calcular $a_1$ , si el segundo bit más bajo está a 1 indica que se ha de calcular $a_2$ , etc., así hasta las componentes que haya programadas. En estos momentos hay hasta $a_5$ .

Los parámetros `param_tansim`, `param_asim1` y `param_asim2` son los análogos a `param_radsim` para las componentes tangencial simétrica y para las dos series de las componentes asimétricas, que en el modelo *radial/tangencial* son las componentes radiales asimétricas ( $c_1, c_2, \dots$ ) y las tangenciales asimétricas ( $d_1, d_2, \dots$ ). El número de componentes asimétricas programado es de 12 para cada serie.

Un elemento **Config\_distorsion** generado mediante la interfaz de Aerotri siempre tendrá a cero los bits correspondientes a las componentes  $a_1, b_1, c_1$  y  $c_2$ , ya que no deben existir sea cual sea la distorsión de la cámara, y el propio concepto de su cálculo junto con los parámetros de orientación de las fotografías no tiene sentido. Para entender por qué esto es así véase el manual de Calibración que se incluye en el instalador de Aerotri.

Además, la interfaz de Aerotri no pregunta por el modelo polinómico o por el modelo de distorsiones asimétricas. Siempre pasará *impar* y *radial/tangencial*. Tampoco pregunta por la variable `condicion_radsim`, y siempre emplea *ortogonal*. En consecuencia tampoco pregunta por `r1`, ya que para *ortogonal* no es de aplicación.

La estructura **Trabajo\_Autocalib** guarda todo el estado de todos los controles relativos a autocalibración. Por lo tanto para que se calculen unas ciertas componentes de distorsión no es suficiente con que los respectivos bits en `param_radsim, ... param_asim2` estén a 1. Es necesario además que estén a 1 las variables `autocalib.existe`, `autocalib.selected.distorsion` y la correspondiente dentro de `autocalib.selected.componentes`; es decir, `autocalib.marcados.radsim`, `autocalib.marcados.tansim` o `autocalib.marcados.asimetricas`. Además `modelo_poli` tiene que ser un modelo reconocido (`CAL_MODPOL_IMPAR` o `CAL_MODPOL_COMPLETO`) así como `modelo_asim` (`CAL_MODASIM_RADTAN` o `CAL_MODASIM_VECTOR`).

Si el resultado de la autocalibración se va a almacenar en un fichero de un formato que incluye como parámetros de distorsión  $K1, K2, K3, P1$  y  $P2$  (llamémosle formato X), en el cálculo de autocalibración mediante Aerotri solamente se pueden incluir las componentes  $a_2, a_3, a_4, d_1$  y  $d_2$ . Si el formato X no permite  $K3$  entonces no se puede incluir  $a_4$ . Además, si se han calculado  $d_1$  y  $d_2$  y el resultado se almacena en el fichero de formato X (el resultado de combinar la interna original con los parámetros calculados en la autocalibración), este fichero se ha de indicar como fichero de interna en nuevo ajuste, y se debe realizar esta

operación hasta que el resultado no varíe. Para una explicación detallada de esto véase la sección «Autocalibración» del manual de Aerotri y dentro de esta los apartados *Parámetros seguros para ficheros de interna no de Aerotri* y *Cálculos iterativos*.

## Valores aproximados

```
typedef struct {
    uint8_t usa_gps;    //0: no; 1: sí (de momento se trata igual que 2); 2: Emplear sólo si
                        //están todos
    bool force_mmcc;    //0: fuerza un ajuste mm.cc., !=0: deja que el programa ajuste
                        //como quiera
    uint16_t hacha;
    uint16_t maximof;   //máximo número de fotos para el ajuste riguroso de todo el
                        //modelo
    uint16_t mult_contra;
} ValoresAproximados;
```

Esta estructura contiene las opciones para el cálculo de valores aproximados, que son muy pocas. Si hay coordenadas gps/ins de los centros de proyección el cálculo es muy sencillo. En ese caso, además del fichero de fotogramas se indicará un fichero de GPS/INS. Además en el parámetro `usa_gps` se pasará 1 para que dichos datos se empleen. Se puede pasar siempre 1, aunque no haya fichero de GPS, ya que si este fichero no existe el parámetro `usa_gps` es ignorado.

El parámetro `force_mmcc` indica, si está a 0, que el ajuste sea puramente mínimo cuadrático. La librería todavía no lo emplea y cálculo es esencialmente mínimo cuadrático siempre.

Los siguientes parámetros se emplean solamente si no hay datos gps/ins para todas las fotos:

<code>hacha</code>	Este parámetro ha de ser el número de puntos en común de un par de fotografías consecutivas tipo. Lo más cómodo es dejar que el programa lo deduzca. Para ello pasar 0.
<code>maximof</code>	El ajuste de valores aproximados no tiene que ser un proceso riguroso, por lo que cuando Aerotri tiene que realizar ajustes de modelos grandes dentro de este proceso no lo hace de modo riguroso para ahorrar tiempo. Si se pasa 0 Aerotri tomará un valor por defecto. Dicho valor es en estos momentos 12.
<code>mult_contra</code>	Pasar 0.

En resumen, se pueden pasar siempre los mismos valores para esta estructura:

```
{1,1,0,0,0}
```

## Precisiones

```
typedef struct{
    float im;           //Coordenadas imagen
    float pXY, pZ;      //Puntos de apoyo
    float gpsXY, gpsZ;  //Coord. gps de los centros de proyección
    float insΩΦ, insK;  //Giros de los centros de proyección
} Precisiones;
```

Esta estructura se corresponde exactamente con los datos pedidos en el recuadro «Precisiones» de la interfaz de Aerotri. Si alguno de los valores no existe se ha de pasar NaN o infinito, lo que se puede conseguir mediante la macro NAN\_F. Por ejemplo, NAN\_F(apriori.insΩΦ). En ese caso serán las funciones de cálculo de la librería las que decidan qué valor de precisión emplear.

## Trabajo\_Opciones

```
typedef struct{
    uint8_t unigiros
    bool reescribe_cpp;
    uint8_t estimador;
    bool bgirok;
    bool fijas;           //Apoyo variable
    bool completo;       //Pasar 1
    uint8_t interp_gps;  //0, no; 1, sólo interpolar; 2, también extrapolar
    bool interp_resgps;
    uint8_t escalaresiduos;
    float residuos_escalas[20];
} Trabajo_Opciones;

#define ATUNIGIROS_Rad      0
#define ATUNIGIROS_Gon     1
#define ATUNIGIROS_Sex     2
#define ESTIMADOR_MMCC     0
#define ESTIMADOR_AEROTRI_2006 3
#define ATTrabajo_GiroINS180_Camara 0
#define ATTrabajo_GiroINS180_Fichero 1
```

Esta estructura incluye varios parámetros heterogéneos que no entran dentro de ninguna otra estructura.

unigiros                    *radianes*, grados *centésimales* (gon) y grados *sexagesimales*. Las unidades de los giros en los ficheros de entrada, que será las

	que se empleen también para los ficheros de salida. Antes solía emplearse grados centesimales, pero desde que se incluyen datos inerciales, que vienen siempre en grados sexagesimales, se emplea siempre estos últimos. Pasar por tanto <b>ATUNIGIROS_Sex</b> .
<code>reescribe_cpp</code>	Indica si el fichero de valores aproximados se ha de reescribir trasformando los puntos aproximadamente al sistema de coordenadas del apoyo y GPS. Cuando se calculan valores aproximados sin ayuda de un fichero de GPS/INS el sistema de coordenadas de los valores aproximados es totalmente arbitrario, con el cero coincidiendo, o casi, con un centro de proyección y la escala de manera que las coordenadas Z de los puntos están en torno a $-1$ . Si se pasa 1 en este parámetro, cuando el cálculo del bloque haya transformado los valores aproximados al sistema del apoyo reescribirá el fichero de valores aproximados, que normalmente tendrá extensión <code>.prm</code> . Aerotri pasa 0 en este parámetro, ya que si el cálculo termina correctamente se genera un fichero <code>.ajs</code> de valores ajustados que está obviamente en el sistema de coordenadas de los puntos de apoyo y reemplaza al <code>.prm</code> de valores aproximados para cálculos futuros.
<code>estimador</code>	Ha de ser un número que selecciona un estimador entre varios predefinidos. De momento solamente hay dos posibles : <i>mm.cc</i> . y <i>Aerotri 2006</i> .
<code>bgirok</code>	indica la manera de interpretar una diferencia de un múltiplo de $90^\circ$ (normalmente $180^\circ$ ) entre la orientación que tiene la fotografía y el valor del giro ins recogido en el fichero. Una posibilidad es interpretar que la cámara está girada en el avión respecto al sistema inercial; la otra consiste en suponer que la fotografía fue girada a posteriori en el ordenador. Para lo primero se pasa <b>ATTrabajo_GiroINS180_Camara</b> y para lo segundo <b>ATTrabajo_GiroINS180_Fichero</b> . Se recomienda no hacer esto último, y por ello el valor por defecto en Aerotri es 0. Para una explicación detallada de la diferencia entre ambas interpretaciones véase la sección correspondiente en el manual de Aerotri.
<code>fijas</code>	Está a 1 si los puntos de apoyo se tratan como observaciones que pueden tener un cierto error o a 0 si se obliga a que sean fijos. Se recomienda pasar 1. En caso de que se obligue a que sean fijos los valores de precisión indicados para el apoyo se ignoran.
<code>completo</code>	Obsoleto hace mucho. Se ha de pasar 1, si bien si se pasa 0 el ajuste del bloque lo primero que hará será ponerlo a 1.

Los parámetros `interp_gps` e `interp_resgps` se refieren a lo mismo. En una pasada puede haber centros de proyección presentes en el fichero GPS/INS pero que no se

encuentran en el fichero de fotogramas; es decir, que no fueron medidos. Aerotri puede aplicar el error sistemático calculado para las coordenadas gps e ins de esa pasada a esos cc.pp. y generar unas coordenadas ajustadas, que se pueden incluir en los ficheros de salida de coordenadas de los cc.pp. Esto es útil por ejemplo cuando se vuela con un 80% de solape con el objetivo de mejorar la calidad de la ortofoto (o por lo que sea). Con este solape se pueden omitir una de cada dos fotos y formar modelos con fotografías alternas, que tienen un 60% de solape. Esto ahorra mucho trabajo ya que hay que medir la mitad de los modelos. Se introduce en el cálculo el fichero de fotogramas con solamente la mitad de las fotografías del vuelo, y Aerotri generará coordenadas ajustadas para todas, sin más que aplicar a las coordenadas en el fichero de GPS/INS de las fotografías que no fueron medidas la corrección debida al error sistemático de dichas coordenadas que Aerotri calcula para cada pasada.

<code>interp_gps</code>	0: No incluir cc.pp. gps/ins intermedios; 1: Incluirlos en los ficheros de salida; 2: Incluir además los cc.pp. anteriores al primero medido de la pasada y los posteriores al último.
<code>interp_resgps</code>	Si está a 1 se corregirá además la coordenada del fichero GPS/INS de un residuo que se calcula como el intermedio entre el residuo de las coordenadas gps/ins del cc.pp. anterior y del siguiente. La idea de este parámetro es corregir errores sistemáticos más complejos que lineales, que se ponen de manifiesto en un cierto sistematismo en los residuos, en lugar de una completa aleatoriedad. Si existe un sistematismo en los residuos (por ejemplo que sean positivos hacia el principio y el final de la pasada, más en los extremos, y negativos en el centro), tiene sentido estimar el residuo que tendría un coordenada GPS promediando los de los puntos anterior y siguiente. Si no existe tal sistematismo la corrección de un supuesto residuo calculado como el promedio de los residuos de los cc.pp. anterior y siguiente solamente añade ruido. Por defecto Aerotri pasa 0; es decir, no corregir de residuo interpolado.
<code>escalaresiduos</code>	Es un parámetro para los ficheros de salida. Refleja lo seleccionado en «Escala para las marcas de los residuos» (véase el manual de Aerotri). Si $0 \leq \text{escalaresiduos} \leq 3$ se trata de una de las series predefinidas en <i>predefined_limites</i> , definido justo antes de la estructura <b>Trabajo_Opciones</b> . Si es $-1$ es otra cualquiera. En cualquier caso los valores límite para cada marca han de estar en <code>residuos_escala</code>
<code>residuos_escala</code>	Serie de números positivos crecientes, cuyo final se indica mediante un NAN_F. Por ejemplo, cuando la función <i>lee_trabajoAT</i> lee un trabajo, si el valor indicado para este parámetros es $n$ , con $0 \leq n \leq 3$ , se ejecuta el siguiente código:

```
trabajo->opciones.escalaresiduos=n;
memcpy_float(trabajo->opciones.residuos_escal,predefined_limites[n],10);
NAN_F(trabajo->opciones.residuos_escal[10]);
```

En realidad, el único punto de la librería en el que se *emplea* el valor de *escalaresiduos* es en *escribe\_trabajoAT*, que escribirá dicho valor y solamente escribirá a continuación los valores de *residuos\_escal* si *escalaresiduos* está fuera del rango de valores reconocidos (e.d., fuera del rango 0-3). Por lo tanto, para pasar a las funciones de la librería de cara al cálculo, los valores límites para las marcas han de estar siempre en *residuos\_escal*, cerrados por un *NAN\_F*.

## Trabajo\_Sistema

```
typedef struct{
    uint nsistema;    //SIS_Rectangular... SIS_Geograficas
    uint elipsoide; //0: Esfera, 1: otro, 2-6: predefinido en ElipsoidesValores
    float ondulacion;
    struct{ Double a,ee;} elipsoideotro;
    TSistema_Tierra tierra;
    TSistema_ValoresProy valoressistemas;
} Trabajo_Sistema;

typedef struct{
    Double N,rho,conv,k;
} TSistema_Tierra;

typedef struct{
    Double kUTM, kLambert;
    Double XUTM,YUTM, XLambert,YLambert;
    ssint Phi0Lambert; //El valor en milésimas de segundo
    bint LatLong;      //0 si el orden es  $\phi,\lambda$ ; 1 si es  $\lambda,\phi$ 
} TSistema_ValoresProy;
```

Esta estructura guarda el sistema de coordenadas seleccionado y los valores indicados para todos los sistemas de coordenadas que Aerotri ofrece. Si la estructura **TrabajoAerotri** en la que se incluye tiene por objeto ser pasada a una de las funciones de cálculo solamente será necesario completar los valores que sean de aplicación al sistema de coordenadas seleccionado. Por ejemplo, si *nsistema* es *SIS\_Rectangular* no hace falta asignar ningún valor más en la estructura.

Lo más importante es que se calcule la aerotriangulación en el mismo sistema de coordenadas que después vaya a emplear el software de restitución, generación de ortofotos, etc. Si dicho software no tiene en cuenta ningún sistema de coordenadas y simplemente

trata las coordenadas como rectangulares se ha de calcular indicando `SIS_Rectangular` o `SIS_Conforme` con `tierra.k = 1`.

<code>nsistema</code>	Uno de los valores definidos en <code>ClaseSistema.h</code> .
<code>elipsoide</code>	Si es 0 se trata de una esfera y el radio se toma como $(\text{tierra.N} + \text{tierra.rho})/2$ . Si es 1 se trata de un elipsoide cualquiera cuyos valores de $a$ y $e^2$ se toman de <code>elipsoideotro</code> . Si es $\geq 2$ sus dimensiones se toman de <code>ElipsoidesValores[elipsoide-2]</code> , definido en <code>ClaseSistema.h</code> .
<code>ondulacion</code>	Ondulación del geoide en la zona. Exactamente, altura sobre la superficie de referencia (elipsoide) de los puntos con Z cero.

Si el sistema de coordenadas es `SIS_Rectangular` no hace falta ningún parámetro más. Cualquier otro sistema empleará en primer lugar el valor de `ondulacion`. Se debe pasar cero en este parámetro salvo que el software que llama a la librería también tenga en cuenta la ondulación del geoide cada vez que trabaje con coordenadas terreno. Este parámetro se eliminó de la interfaz de `Aerotri` porque los usuarios se preocupaban de buscar el valor de ondulación del geoide en la zona cuando en realidad debían indicar cero, ya que los distintos software de restitución nunca tenía en cuenta este valor.

El elipsoide indicado en `elipsoide` será de aplicación si `nsistema` es distinto de `SIS_Conforme`. Para este último sus parámetros se toman de `tierra`:

<code>N</code>	Radio de curvatura mayor; esto es, según la dirección E-O.
<code>rho</code>	Radio de curvatura menor; esto es, según la dirección N-S.
<code>conv</code>	Ángulo que forma el eje Y con la dirección de curvatura menor; es decir, con el Norte. Positivo si hacia el Este.
<code>k</code>	Factor de escala de la proyección en la zona. $<1$ si la distancia de dos puntos sobre la proyección es menor que sobre el elipsoide.

El sistema `SIS_Conforme` es un sistema genérico que permite imitar localmente un sistema conforme cualquiera con precisión suficiente para una aerotriangulación salvo para grandes extensiones de terreno. Lo más importante es indicar correctamente el valor de `k`. Si el software de resitución no va a tener en cuenta el sistema de coordenadas se puede calcular la aerotriangulación indicando `SIS_Conforme` para que al menos se tenga en cuenta la esfericidad de la Tierra, acordándose de pasar `k=1`.

Los valores de `N` y `rho` no tienen mucha importancia. `conv` sólo importa si `N  $\neq$  rho`. La mayoría de las veces es suficiente `N = rho = 6376000`.

Para extensiones de miles de kilómetros cuadrados la simulación de un sistema conforme cualquiera que permite `SIS_Conforme` puede no ser suficiente. No obstante, para extensiones tan grandes la tendencia debería ser a que los ficheros estén directamente en coordenadas geográficas.



## TSistema\_ValoresProy

Guarda los valores seleccionados para todas las proyecciones que soporta la librería de Aerotri. En breve se añadirá la estereográfica oblicua y probablemente ya no se añada ninguna más.

kUTM, kLambert	Factor de escala en el meridiano o paralelo central.
XUTM, YUTM, XLambert, YLambert	Coordenadas del punto central de la proyección. Para la proyección UTM la coordenada YUTM es la de los puntos del Ecuador.
Phi0Lambert	Para la proyección Lambert. La latitud del paralelo central en milésimas de segundo.
LatLong	Para geográficas. 0 si el orden en los ficheros es $\varphi, \lambda$ ; 1 si es $\lambda, \varphi$ .

## AjbloqueConfigSalida

Esta es una estructura bastante grande que se corresponde con el contenido de la pestaña «Ficheros de salida» de Aerotri, salvo por la escala para las marcas de los residuos que se incluye en **Trabajo\_Opciones**. Por ello no se hará una descripción detallada de cada campo sino que se dirá con qué control de la GUI se corresponde y si acaso se hará una referencia al manual de Aerotri.



```

typedef struct{
    struct{
        /* ... */
    } ficheros;
    struct{
        /* ... */
    } general;
    struct{
        /* ... */
    } pares_pasadas;
    struct{
        /* ... */
    } fich_Digi;
    struct{
        /* ... */
    } fich_Imaster;
    struct{
        /* ... */
    } fich_inf;
    struct{
        /* ... */
    } fich_std;
    struct{
        /* ... */
    } fich_pdf;
    struct{
        /* ... */
    } fich_dibujo;
} AjbloqueConfigSalida;

```

La estructura `ficheros` consta de campos de un solo bit, según puede verse en `clases_mainabsoluta.h`, que se corresponden con cada uno de los *check box* que permiten seleccionar un fichero o un grupo de ficheros. Los dos primeros bits: `res` e `inc` no se corresponden con ningún *check box* porque son ficheros que ya no se generan; los generaban las primeras versiones de Aerotri.

La estructura `general` se corresponde con el recuadro «Opciones generales», y junto con los comentarios en el fichero `.h` no hace falta más explicación. Nótese que si bien la interfaz de Aerotri, en la elección del número de decimales a mostrar para los residuos, si se quiere un número fijo de cifras significativas solamente da la opción de dos cifras significativas, la estructura **AjbloqueDecimales** `decres`, que es donde se pasan esos valores, permite cualesquiera otros valores.

La estructura `pares_pasadas` se corresponde con «Pares» dentro de «Opciones generales». Se explica en la sección «Configuración de la salida → Pares» en el manual de Aerotri. `brev` se corresponde con «Orientar las pasadas en el mismo sentido» y significa que si, por ejemplo, el avión voló en dirección E-O en una pasada y O-E en la siguiente, los pares se generarán de manera que la fotografía izquierda sea siempre la del Este, o siempre la del Oeste, de manera que al abrir un par el Norte quede siempre hacia arriba o siempre hacia abajo, igual en ambas pasadas.

`fich_Digi` y `fich_Imaster` se corresponden con las opciones para los grupos de ficheros de salida de Digi e Image Master. En ambos casos `files` son flags que indican los ficheros que se han de generar, según se indica en el comentario al mismo en [clases\\_mai-nabsoluta.h](#). Las rutas `rutarel` y `rutaf` son las dos rutas por las que se pregunta en el cuadro de diálogo, en el mismo orden en que se piden. Los valores `px`, `py` de `fich_Digi` son las coordenadas del punto principal para escribir en los ficheros `.rel`. Son simplemente informativas y la interfaz de Aerotri no los pide y pasa siempre (0, 0).

`fich_dibujo` pasa opciones para el gráfico del conjunto y el gráfico de los residuos y distorsiones. De momento los campos `hwnd` y `ConfigDibujo` se ignoran, mientras que los arrays de nombres de ficheros `configs_main` y `configs_residuos`, que han de estar cada uno terminados por un puntero `NULL`, se establecen siempre a unos valores fijos en la interfaz de Aerotri, correspondientes a nombres de ficheros que se instalan con el programa, sin que el usuario pueda seleccionar otros, si bien se mira primero si existe una versión de ese fichero en la carpeta de datos del programa específica del usuario, por lo que si no el nombre, sí el contenido se puede modificar en esa copia local. `configs_main` y `configs_residuos` pueden ser `NULL`. Excepcionalmente, estos arrays de nombres de ficheros no se guardan en el fichero de trabajo.

### `fich_inf` y `fich_std`

```
struct {
    unsigned infR : 16;
    unsigned infP : 16;
} fich_inf;

#define AJ_INFR_Fotogramas 1
...
#define AJ_INFR_Elevados 0x20
#define AJ_INFP_ParCentros 1
...
#define AJ_INFP_PrecPuntosTodos 0x100
```

Se corresponde esta estructura con los checkbox de la ventana que se muestra al pulsar el botón a la derecha de «Texto (.inf)» y «Html (.html)». Los bits de `infR` e `infP` indican

si uno u otro bloque de información ha de incluirse en dichos ficheros. `infR` para los residuos e `infP` para los valores ajustados y las precisiones. Las definiciones de macros que comienzan por `AJ_INFR_` permiten discernir el significado de cada bit de `infR`, mientras que las que comienzan por `AJ_INFP_` se corresponden con los bits de `infP`. En esta última las macros correspondientes a valores ajustados contienen la abreviatura `Par` (de parámetros) mientras que las correspondientes a precisiones contienen en su nombre `Prec`.

`AJ_INFP_PrecCentros`

indica si se han de mostrar las precisiones de los centros de proyección, mientras que `AJ_INFP_PrecCentrosTodos` indica si se ha de mostrar solamente una muestra o todos. Análogamente para los puntos. Las macros terminadas en `Grupo` hacen referencia a los parámetros de desviación (error sistemático) de los grupos (normalmente las pasadas) `gps/ins`.

```
struct{
    unsigned stdR :16;
    unsigned stdP :16;
} fich_std;

#define AJ_STDR_Distribucion 1
...
#define AJ_STDR_GpsIns      8
#define AJ_STDP_PrecCentros 1
...
#define AJ_STDP_CorrCentrosTodos 0x80
```

Análogamente al fichero de información esta estructura contiene dos variables cuyos bits indican si tal o cual información se incluye en el fichero de estadísticas. La macro `AJ_STDR_Distribucion` hace referencia a la distribución de los residuos, mientras que el resto de las macros `AJ_STDR_` hacen referencia a las redundancias parciales de cada conjunto de observaciones. La escritura de estas últimas aún no está programada por lo que los correspondientes bits todavía no tienen efecto.

El bit `AJ_STDP_CorrCentrosTodos` quiere decir que se muestren las correlaciones de los parámetros de orientación entre centros de proyección con puntos comunes. Tampoco está programado todavía.

## Trabajo\_Salida

```
typedef struct{
    bint log_ajuste; //Fichero de log para el ajuste
    bint log_pdfs;   //Mantener los logs de generación de los pdfs (de TeX)
    u16int log_level; //Nivel de log para ajbloque y relativa
    bint muestra_html; //Mostrar html tras el cálculo
} Trabajo_Salida;
```

Esta estructura contiene unas pocas variables relativas a los ficheros de salida que no se incluyen en `AjbloqueConfigSalida`. Las variables `log_pdfs` y `muestra_html` no se usan en las funciones de cálculo de la librería de Aerotri ya que corresponden a acciones a llevar a cabo por la GUI. En la GUI de Aerotri, si `log_pdfs` está a 1 se omite la eliminación de ciertos ficheros `.log` intermedios.

`log_ajuste` indica si la GUI de Aerotri creará un fichero de log para pasar a la función *calcula\_absoluta* para que lo escriba con los pasos que el ajuste del bloque va siguiendo. Puesto que dicho fichero es un parámetro que se pasa a *calcula\_absoluta*, y si no se quiere lo que hay que hacer es pasar `NULL`, la variable `log_ajuste` no interviene en las funciones de cálculo de la librería (el trabajo la guarda como una indicación de lo que ha de hacer la GUI). `log_level` indica el nivel de información mostrado en el log. Para *calcula\_relativa* la información a glosar se incluye en el fichero `.pro` de proceso, que se genera siempre.

## Funciones de cálculo

```
int calcula_trabajoAT(Handler hwnd_main, int log_code, TrabajoAerotri *tr,
    char16_t* mensaje, uint8 idioma);
```

```
int calcula_relativa(Handler hwnd_main, int log_code,
    const struct strRelativaParams *params, InfoMalos *pinfomalos,
    char16_t* mensaje, uint8 idioma);
```

```
int calcula_absoluta(Handler hwnd_main, int log_code, Buffer_to *log_file,
    const struct strAbsolutaCallData *params, FicheroBnf *pinfo,
    char16_t* mensaje, uint8 idioma);
```

Los parámetros `hwnd_main`, `log_code`, `mensaje` e `idioma`, comunes a las tres funciones, se explican en `aerotri_trabajos.h`, con motivo de la explicación de *calcula\_trabajoAT*. El parámetro `params` ha de ser el resultado de una llamada con éxito a *prepara\_trabajo\_relativa* o *prepara\_trabajo\_absoluta*.

El parámetro `log_file` en *calcula\_absoluta* es el fichero de log, si se quiere uno; en caso contrario se ha de pasar `NULL`. La información de este fichero no tiene mucho interés y nadie cambia la opción por defecto que es pasar `NULL`, por lo que no vamos a extendernos aquí en explicar la estructura `Buffer_to`. Si se pasa un puntero a dicha estructura el fichero ha de estar ya abierto, *calcula\_absoluta* escribe en él y no lo cierra.

Falta por tanto por explicar los parámetros `pinfomalos` y `pinfo`. Ambos son punteros a estructuras declaradas en ficheros de cabecera incluidos por `aerotri_trabajos.h` que las

funciones de cálculo han de rellenar. Si no se quiere esa información se ha de pasar **NULL** en dichos punteros.

## InfoMalos

Esta estructura se rellena por *calcula\_relativa* en caso de que el valor devuelto sea 10 u 11, lo que indica un error en el cálculo. Véase *mainrelativa.h*. Si lo que se quiere es mostrar al usuario información acerca de posibles pares de fotos o puntos malos no es necesario emplear esta estructura. Esa información ya se presenta en el archivo .pro que se genera siempre en el cálculo de la relativa. Aerotri, por ejemplo, no emplea esta estructura y siempre pasa **NULL** en este parámetro. La información devuelta en esta estructura sirve para un correlador, para eliminar puntos malos o volver a correlar puntos nuevos, y tras ello volver a llamar a la librería. De esta manera el proceso de remeida y recálculo es transparente para el usuario, que no es consciente de que en un primer momento el cálculo dio error.

Si relamente se pasa un puntero a una estructura (en lugar de pasar **NULL**) *calcula\_relativa* la rellenará con punteros a memoria reservada que han de liberarse mediante una llamada a *free\_plist* como sigue:

```
free_plist(informalos.plist);
```

Este código puede ejecutarse aunque la estructura no se haya rellenado, por lo que lo más cómodo es ejecutarlo siempre tras una llamada a *calcula\_relativa*.

```
typedef struct {
    PLIST plist;
    UnPaso paresmalos[10];
    Ajuste_con_PuntoMalo* ajustesmalos;
    char* nombresf;
    char* nombresp;
    uint* fotos;
    uint* puntos;
} InfoMalos;
```

El proceso de cálculo de valores aproximados, en ausencia de datos gps/ins para los centros de proyección, consiste en la unión de fotogramas para formar modelos. Después estos modelos se van uniendo entre sí o con fotogramas que todavía queden sueltos para ir formando modelos cada vez mas grandes. Tras el último paso el resultado es un modelo que contiene todas las fotos del trabajo. Cada uno de estos pasos une dos elementos y da como resultado un elemento. Los elementos pueden ser fotogramas o modelos ya existentes. Es la información que se muestra en el fichero de proceso.

Por ejemplo, un paso puede consistir en la unión de las fotografías 002435 y 002437 para formar el modelo 23. Otro paso puede ser la unión del modelo 23 con la fotografía

002502, cuyo resultado es otra vez el modelo 23. Otro paso puede ser la unión de los modelos 65 y 23 para dar como resultado el modelo 65; es decir, el modelo 65 se ha *comido* al 23. La información de cada paso se codifica en una estructura de tipo **UnPaso**:

```
typedef struct{
    uint elem_r;      //n de modelo resultado
    uint elem1;       //Si <0 es fotograma, si>0 un modelo.
    uint elem2;
    uint npuntos;     //número de puntos en común
    uint ncentros;    //número de fotos del modelo resultado
    Resultado ajuste;
} UnPaso;
```

Tanto los modelos como las fotografías se identifican con un número. Los números de fotografías empiezan en 0 y los de modelos en 1 (no existe el modelo 0). `elem_r` es el número del modelo del resultado, ya que el resultado de un paso siempre es un modelo. `elem1` y `elem2` son los dos elementos que se unen. Si el número de tipo `uint` tiene el bit más alto a 0 se trata de un número de modelo; si tiene el bit más alto a 1 se trata de un número de fotograma, codificado en complemento a unos; es lo que en los comentarios se indica como <0. Se pueden emplear las macros `ispos_uint` e `isneg_uint` para distinguir unos de otros. La macro `ispos_uint` también se evalúa a *true* si el número es cero:

```
/* paso is of type UnPaso* */
if(ispos_uint(paso->elem1)){ //Se trata de un modelo
    uint m=paso->elem1;
    /* ... */
}else{ //se trata de una foto
    uint f=~pasos->elem1;
    /* ... */
}
```

Cuando se unan un modelo y una fotografía siempre será `elem1` el modelo y `elem2` la fotografía.

Cuando un ajuste de valores aproximados da error hay al menos un paso que dio error. Estos pasos se almacenan en el array `ajustesmalos` dentro de `infomalos`. El primer elemento del array con `.najuste=EOUIA` indica el final. Véanse definiciones y comentarios en `mainrelativa.h`. El propio elemento con `.najuste=EOUIA` ya no contiene nada.

El contenido de **Ajuste\_con\_PuntoMalo** se define y explica en `mainrelativa.h`. Da información acerca del punto peor del ajuste, que es el posible punto malo. `ajuste.p_peor` es el número del punto peor y `nast` una variable que indica cómo de verosímil es que el punto realmente sea malo, mediante un número que va de cero a cuatro. Esto se corresponde exactamente con lo mostrado en el archivo de proceso para cada ajuste malo.

La estructura `infomalos` también contiene un array de pares de fotos que se consideran que pueden ser malos. Un par con `ajuste.vv=0` indica el final. Puesto que el array se define como `UnPaso paresmalos[10]`, este puede contener entre cero y nueve posibles pares malos. Cómo de probable es que el par sea malo depende de `ajuste.prop_peor`, `ajuste.vvmax` y `npuntos`, y depende mucho del correlador empleado, de modo que unos valores pueden ser indicativos de un punto malo para medidas procedentes de un cierto algoritmo de correlación pero no para otros. Por otra parte debe recordarse que en pares con solamente seis puntos en común el punto malo no puede deducirse salvo que supongamos *a priori* una posición relativa aproximada para los fotogramas (por ejemplo, aproximadamente en caso normal). En pares con seis puntos en común el punto `p_peor` muchas veces no es el punto malo.

## Identificación de fotografías y puntos

Se ha visto que tanto las fotografías como los puntos se identifican mediante un número. Para asociar un número de foto o de punto a su nombre, la estructura `InfoMalos` contiene los arrays `nombresf`, `fotos`, `nombresp` y `puntos`, para cuya comprensión y empleo son suficientes los comentarios adyacentes a su definición en el fichero `mainrelativa.h`.

## FicheroBnf

```
typedef struct {
    PLIST plist;
    NamedPointer* punteros;
    Namedn* enes;
    BnfVarios *varios;
} FicheroBnf;
```

La función `calcula_absoluta` recibe un puntero a esta estructura para rellenarlo con resultados del cálculo. Es la misma información que se escribe en el fichero `bnf`. Si no se quiere recibir esta información se ha de pasar `NULL` en `pinfo`. Si en lugar de esto se pasa un puntero a una estructura para que se rellene, posteriormente ha de liberarse la memoria reservada mediante

```
free_plist(informalos.plist);
```

al igual que para `InfoMalos` `*pinfomalos` en `calcula_relativa`. Esta llamada no hará nada si debido a un error en el ajuste la estructura no se llegó a rellenar.

## BnfVarios

```
typedef struct{
    ...
    struct{
        uint ncp, npm, npy, ngruposgps,ngps, ngruposins,nins, ngruposcalib;
    } ncalc;
    Precisiones apriori; //Los valores para magnitudes angulares,
    Precisiones aposteriori; //INSΩΦ e INSK, están en radianes
    float σ_aposteriori; /*En relación a los valores a priori*/
    struct{
        float foto, apoyo, gps, ins;
    } redun_parciales;
    uint nobs;
    uint nincog;
    DecimalesAj decj;
} BnfVarios;
```

Esta es una estructura con información variada. Los elementos anteriores a `ncalc`, que van de `fichero` a `estimador`, no se muestran porque coinciden con campos de la estructura **TrabajoAerotri** que ya se han explicado.

`ncalc` contiene el número de elementos calculados, para distintos elementos:

<code>ncp</code>	Centros de proyección (cc.pp.); e.d., fotografías
<code>npm</code>	Puntos, incluyendo los de apoyo
<code>npy</code>	Puntos de apoyo
<code>ngruposgps</code>	Conjuntos con datos gps
<code>ngps</code>	Número de observaciones de cc.pp. gps o ins. En la práctica son siempre de gps y pueden ser además de ins
<code>ngruposins</code>	Conjuntos con datos ins (giros)
<code>nins</code>	Número de cc.pp. ins
<code>ngruposcalib</code>	0 si no hay datos de calibración, 1 si hay

`apriori` también se corresponde con datos de entrada, y `aposteriori` son las desviaciones típicas que **Aerotri** deduce para cada conjunto de datos a raíz del ajuste. `σ_aposteriori` es un valor global que indica la precisión de los datos en relación a lo especificado a `priori`. Si es menor que 1 es que del ajuste se infiere que la precisión es mejor que la indicada en los datos de entrada; si es mayor que 1, que es peor.

Las redundancias parciales permiten deducir si hay datos del tipo en cuestión: si el valor es 0 no hay datos de ese tipo. Nótese que la indicación de un fichero de entrada no implica que contenga datos que vayan a entrar en el cálculo. Por ejemplo, si por equivocación se indica un fichero de GPS/INS de otro trabajo, en el que ningún nombre de centro



de proyección coincide con los nombres de las fotos medidas, no habrá datos gps en el cálculo.

foto	De las fotocoordenadas. Nunca puede ser cero
apoyo	De las coordenadas de los puntos de apoyo
gps	De las observaciones gps de los cc.pp.
ins	De las observaciones ins de los cc.pp.

nobs y nincog son respectivamente el número total de observaciones y de incógnitas del ajuste. En un ajuste mínimo cuadrático la suma de las redundancias parciales tiene que ser igual a la resta de estos dos valores. En un ajuste robusto puede variar ligeramente.

La estructura **DecimalesAj** decj contiene el número de cifras decimales con que se han de mostrar los valores ajustados, deducidos a partir de las precisiones *a priori* proporcionadas. Los valores que contiene son siempre positivos o cero, indicando el número de decimales tras la coma (y no, por tanto, el número de cifras significativas).

## Namedn

```
typedef struct {  
    const char* name;  
    uint n;  
} Namedn;
```

Este es un simple array de pares de nombre/valor. En el fichero `clases_infobinario.h` el array constante `bnf_nombres_enteros` contiene los posibles valores de name.

"n fotografias"	Número de fotografias en el fichero de fotografias, sin contar los vacíos
"n CentrosProy"	Número de centros de proyección en el fichero de valores aproximados que se corresponden con fotografias del fichero de fotografias. Normalmente coincide con el número de cc.pp. calculados ( <code>varios-&gt;ncalc.ncp</code> ), pero este puede ser menor debido a centros que se hayan eliminado del cálculo por no tener suficientes puntos.
"n PuntosM"	Número de puntos en el fichero de valores aproximados que se corresponden con puntos en el fichero de fotografias. Suele coincidir con el número de puntos calculados pero este último puede ser menor por algún punto eliminado por no aparecer en al menos dos fotografias.
"n puntosA"	Número de puntos de apoyo en el fichero de apoyo.
"n puntosC"	Número de puntos de control en el fichero de apoyo.
"n GruposNAV"	Número de grupos gps/ins
"n gpss"	Número de obervaciones gps de cc.pp.

"n ins"                    Número de observaciones ins de los cc.pp.  
"n observaciones" Número de fotocoordenadas medidas que entran en el cálculo

## NamedPointer

```
typedef struct {  
    uint tipo;  
    const char* name;  
    uint n;           //Número de elementos  
    void* p;  
} NamedPointer;
```

p	Puntero a la dirección de memoria que almacena el array de elementos de tipo tipo.
tipo	Tipo de elementos del array p. Los posibles tipos están declarados en el mismo fichero que la estructura.
n	Número de elementos del array.
name	Nombre que identifica el array almacenado.

El tamaño de un elemento de cada tipo está definido en `NamedPointersSizeof`, de manera que es

`NamedPointersSizeof[tipo]`.

y el tamaño total del array es el número anterior multiplicado por n. Los posibles nombres son los que se listan en `bnf_nombres_punteros`.

Aerotri usa la información devuelta en esta estructura para mostrar los residuos en un cuadro, ordenados de mayor a menor por defecto. Para hacer esto mismo el programador puede analizar la estructura y organizar una tabla como desee, o bien emplear las funciones declaradas en `infobinario.h`. También se puede emplear el visor de Aerotri sin más que ejecutar `EditordeDatos.exe` pasando como argumento el nombre del fichero con extensión `.bnf` generado en el cálculo. (El nombre `EditordeDatos` no es adecuado, ya que no se edita nada, pero ha quedado así por razones históricas).

# Infobinario

Esta librería consta de los ficheros `infobinario.h`, `clases_infobinario.h` y algún fichero de cabecera más incluido por estos. Para compilar hay que incluir

```
#include "infobinario.h"
```

y hacer referencia a `infobinario.lib`, y la dll con el código ejecutable es `infobinario.dll`.

La librería declara la estructura **FicheroBnf** y proporciona una función que lee un fichero binario de Aerotri, de extensión .bnf, rellendo una estructura de tipo **FicheroBnf**; una función que escribe el fichero a partir de la estructura, y funciones que rellenan tablas de residuos a partir de la estructura.

La descripción de la estructura se encuentra al final de capítulo anterior.

## Extraer los residuos de las fotocoordenadas

El número total de puntos medidos es el entero indicado en "**n observaciones**"; es decir, el campo `.n` del elemento del array `enes` cuyo campo `.name` apunta a la cadena "**n observaciones**". Los residuos se encuentran en el array de nombre "**L puntosf**", de tipo float; es decir, `tipo==TIPOBNF_F`. Si llamamos `L` a este array los residuos de la observación `i` se encuentran en `L[2*i]` y `L[2*i+1]`. Son los residuos de la fotocoordenada  $x$  y la fotocoordenada  $y$  respectivamente. Para saber a qué fotograma y punto corresponde esa observación se ha de mirar el array "**observaciones**", de tipo `int4` (`TIPOBNF_U4`). El elemento `observaciones[i]` contendrá la siguiente información:

<code>n1</code>	Número de punto
<code>n2</code>	Número de punto de apoyo. -1 si no es punto de apoyo
<code>n3</code>	Número de centro de proyección (fotograma)
<code>n4</code>	Número de punto dentro del fotograma

El nombre del punto se puede extraer mediante `PuntosModelo[n1].nom`.

El nombre del fotograma se puede extraer mediante `CentrosProy[n3].nom`.

`PuntosModelo` y `CentrosProy` son los arrays homónimos; es decir, de nombre "**PuntosModelo**" y "**CentrosProy**" respectivamente.

## Extraer los residuos de los puntos de apoyo

Hay el entero de nombre "**n puntosA**" puntos de apoyo. Sus residuos se encuentran en el array de nombre "**L puntosA**". Si llamamos `L` a este array los residuos en  $X, Y$  y  $Z$  del punto de apoyo `i` se encuentran en `L[3*i]`, `L[3*i+1]` y `L[3*i+2]`.

Algún punto de apoyo puede no haber entrado en el cálculo. Para ello se mira el array "**puntosA\_a\_PuntosM**". Sea `pApM` ese array. Si `pApM[i]==-1` entonces el punto de apoyo no entró en el cálculo y se debe ignorar.

Sea `puntosApoyo` el array homónimo. El nombre del punto de apoyo `i` es `puntosApoyo[i].nom`.

## Extraer los residuos de los puntos de control

Hay (el entero de nombre) "**n puntosC**" puntos de control. Sus residuos se encuentran en el array de nombre "**L puntosC**", análogo al array "**L puntosA**" para los puntos de apoyo.

Algún punto de control puede no haber entrado en el cálculo. Para ello se mira el array "**puntosC\_a\_PuntosM**". Sea  $pCpM$  ese array. Si  $pCpM[i] == -1$  entonces el punto de control no entró en el cálculo y se debe ignorar.

Sea `puntosControl` el array homónimo. El nombre del punto de control  $i$  es `puntosControl[i].nom`.

## Extraer los residuos de GPS e INS

Hay un total de (el entero de nombre) "**n gpss**" observaciones de gps y/o ins de los centros de proyección. El array "**gpss**" contiene dichas observaciones.

<code>gpss[i].nom</code>	El nombre del centro de proyección de la observación $i$
<code>gpss[i].bg</code>	Si es <i>true</i> hay observación de gps (X, Y, Z); si es <i>false</i> , no
<code>gpss[i].bi</code>	Si es <i>true</i> hay observación de ins ( $\Omega$ , $\Phi$ , K); si es <i>false</i> , no

Sus residuos se encuentran en los arrays de nombre "**L gpss**" y "**L inss**". Sean  $Lg$  y  $Li$  dichos arrays. Los residuos de la observación  $i$  estarán en  $Lg[3*i]$ ,  $Lg[3*i+1]$ ,  $Lg[3*i+2]$ ;  $Li[3*i]$ ,  $Li[3*i+1]$  y  $Li[3*i+2]$ .

La función *calcula\_absoluta* de la librería de Aerotri pone a *false* los campos `bg` y `bi` de las observaciones que no se corresponden con un fotograma que entre en el cálculo de la aerotriangulación.

## Residuos normalizados

Para cada residuo se puede calcular el cociente entre su valor y lo que cabría esperar de él en el ajuste. Dichos cocientes se almacenan en los arrays "**Lnorm puntosf**", "**Lnorm puntosA**", etc. Los valores almacenados son positivos o negativos según sea el residuo original.

## Las funciones que rellenan tablas

Son las siguientes :

```
int rellena_res_porfotos(BnfTabla_ResporFoto *tabla, ...
int rellena_res_porpuntos(BnfTabla_ResporPunto *tabla, ...
int rellena_res_apoyo(BnfTabla_ResApoyo *tabla, ...
int rellena_res_gpsins(BnfTabla_ResGPS *tabla, ...
```

Como puede verse el primer argumento de todas ellas es una tabla. Dicha tabla es rellena por la función, para lo cual ha de reservar memoria. Posteriormente las filas de la tabla han de liberarse mediante la función *free\_filas*. La definición de las tablas se encuentra al final del fichero *clases\_infobinario.h*:

```
typedef struct{BnfFila_ResporFoto *filas; uint nfilas;} BnfTabla_ResporFoto;
typedef struct{BnfFila_ResporPunto *filas; uint nfilas;} BnfTabla_ResporPunto;
typedef struct{BnfFila_ResApoyo *filas; uint nfilas;} BnfTabla_ResApoyo;
typedef struct{BnfFila_ResGPS *filas; uint nfilas;} BnfTabla_ResGPS;
```

Si hemos pasado a una de esas funciones un puntero a una tabla de nombre *tabla* deberemos liberar las filas así:

```
free_filas(tabla.filas);
```

En las funciones *rellena\_res\_porfotos* / *rellena\_res\_porpuntos* hay un parámetro de nombre *llena\_nombres*. Si está a *false* se rellenan las filas como las muestra el visor de resultados binarios de Aerotri, dejando en blanco el primer elemento de la fila para las filas que no sean la cabecera de una foto / punto. Si está a *true* se completará ese elemento con el nombre de la foto / punto. Lo mejor para entender estas funciones es llamarlas y observar las filas creadas en la tabla.

Los parámetros *dec*, o bien *decgps* y *decins* indican el número de decimales con que se formaterán los residuos, según la explicación del propio fichero *infobinario.h*, justo encima de la declaración de las funciones. El valor *fccpi* es el multiplicador que pasa de radianes a las unidades en que queremos que se muestren los valores angulares. Si por ejemplo queremos que se muestren en grados sexagesimales el valor a pasar en *fccpi* será  $180/\pi = 57,29578$ .

El parámetro *fixas* es el del mismo nombre en la estructura *Trabajo\_Opciones*. Si es *false* los puntos de apoyo se han mantenido fijos y no hay por tanto residuos de los puntos de apoyo. Podrá haber residuos de los puntos de control, ya que esto último no tiene nada que ver con que los puntos de apoyo sean fijos o no.

Los demás parámetros son enteros o punteros de la estructura **FicheroBnf**, según se indica a continuación:

Enteros:

<i>ncp</i>	"n PuntosM "	<i>npm</i>	"n PuntosM"
<i>ntot</i>	"n observaciones "	<i>npv</i>	"n puntosA"
<i>npc</i>	"n puntosC "	<i>ngrupos</i>	"n GruposNAV"

Punteros a arrays:

<i>orden_cp</i>	"Orden CentrosProy "	<i>p_cp_acum</i>	"Acumulado CentrosProy"
-----------------	----------------------	------------------	-------------------------

mafp	"observaciones "	centros	"CentrosProy"
puntosM	"PuntosModelo "	pm_ppios	"Ppios PuntoM_en_CP"
puntosA	"puntosApoyo "	orden_pm	"Orden PuntosM"
pA_a_pM	"puntosA_a_PuntosM "	puntosC	"puntosControl"
pC_a_pM	"puntosC_a_PuntosM "	grupos	"GruposNAV"
grupos_acum	"Acumulado GruposNAV "	gpss	"gpss"
L	"L puntosf "	L_norm	"Lnorm puntosf"
Lp	"L puntosA "	Lp_norm	"Lnorm puntosA"
Lpc	"L puntosC "	Lpc_norm	"Lnorm puntosC"
Lgps	"L gpss "	Lgps_norm	"Lnorm gpss"
Lins	"L inss "	Lins_norm	"Lnorm inss"

## Las estructuras fila

Los nombres de los campos de estas estructuras dan cuenta de su significado de manera clara (v. el fichero [clases\\_infobinario.h](#)). Los que comienzan por `v_` son columnas que el visor de ficheros `bnf` de Aerotri no muestra al usuario. Las columnas `_resx`, etc. son los valores de los residuos, como un número **float**, mientras que `f2_resx`, etc. es lo mismo formateado como texto. Las columnas cuyo nombre contiene `res_norm` son residuos normalizados, en valor absoluto (es decir, siempre positivos).

Las columnas que comienzan por `v_orden_` contienen enteros que sirven para ordenar las filas según esa columna, pudiendo ordenarse por el orden por defecto, por residuo de mayor a menor o por residuo normalizado.

En las filas que rellena la función *rellena\_res\_apoyo* se intercala una fila vacía entre los puntos de apoyo y los puntos de control. En las filas de las funciones *rellena\_res\_porfotos* y *rellena\_res\_porpuntos* hay una fila de cabecera para cada foto o punto respectivamente seguida de las filas con los residuos de los puntos de la foto / de ese punto en las distintas fotos. Para entender esto bien no hay más que llamar a estas funciones y observar las filas devueltas, o bien observar las filas mostradas por el visor de ficheros `.bnf` de Aerotri al abrir uno de ellos.

## Distorsión

La librería *distorsión* permite la lectura y escritura de ficheros de interna de Aerotri y su aplicación a fotocoordenadas medidas, en uno y otro sentido. El fichero de cabecera es [distorsion.h](#), la referencia para enlazar es [distorsion.lib](#) y la dll [distorsion.dll](#). Si se incluye

[distorsion.h](#) además de [aerotri\\_trabajos.h](#) o [infobinario.h](#) ha de hacerse después de estos ficheros. Los comentarios del fichero .h bastan para el uso de esta librería.

## Lectura y escritura de ficheros

La librería reconoce los siguientes formatos. No todos se pueden leer ni todos se pueden escribir. Los que se pueden leer se indican con //R, los que se pueden escribir con //W, y con //R/W los que la librería es capaz de leer y escribir.

```
#define CAMFORMAT_Aerotri    0        //R/W
#define CAMFORMAT_ini        1        //R/W
#define CAMFORMAT_cmr        2        //R/W
#define CAMFORMAT_RapidCal   3        //R
#define CAMFORMAT_Digi       4        //W
#define CAMFORMAT_Agisfot    5        //W
```

La lectura del fichero se hace a través de la función *lee\_ficheroint* :

```
int lee_ficheroint(const char16_t *ficheroint, uint inm,
                  OrientacionInterna* interna,
                  char16_t* mensaje, uint idioma);
```

El parámetro *interna* apunta a una estructura que se llenará si la lectura tiene éxito. Véase el fichero [distorsion.h](#) para los posibles valores devueltos y la explicación de los parámetros de la función.

La escritura del fichero se hace mediante una función específica para cada tipo de fichero. Véase el fichero [distorsion.h](#) para las distintas posibilidades. La función que escribe el fichero en el formato de Aerotri recibe un parámetro de tipo **const OrientacionInterna \***; las demás reciben **const InternaKPB \***. Una estructura de tipo **OrientacionInterna** se puede transformar en **InternaKBP** y viceversa. Se explica más abajo.

Lo primero que hace la función *lee\_ficheroint* es inicializar la estructura *interna*, antes incluso de intentar abrir el fichero. Por tanto cualquier contenido que tenga se destruirá. El contenido de la estructura *interna*, es decir, la memoria a la cual los punteros de dicha estructura apuntan, ha de liberarse posteriormente mediante *free\_interna*, incluso aunque la función devuelva algún código de error, pues pudo haberse rellenado parcialmente. Si el fichero no se llega a abrir no es necesario; no obstante no hace daño llamar siempre a *free\_interna* y es lo mas fácil y seguro.

## OrientacionInterna

```
typedef struct{
    KeyVal *info;
    uint flags;
    Lineal2D_fl afin_1;
    Valores_Interna valI;
    Distorsion distorsion;
} OrientacionInterna;
```

La estructura **OrientacionInterna** representa una orientación interna de Aerotri. El array de **KeyVal** contiene información sobre la cámara o la calibración. De momento hay definidas las siguientes claves (valores de key).

```
#define CAL_KV_END      0          //Final del array
#define CAL_KV_EMPTY ((uintptr_t)-1) //Par clave/valor vacío
#define CAL_KV_minx     1
#define CAL_KV_maxx     2
#define CAL_KV_miny     3
#define CAL_KV_maxy     4
```

Para estas claves el valor es **float**; es decir, la unión **val** contiene el float **val.f1**. Si el valor de la clave es **CAL\_KV\_EMPTY** el par clave/valor está vacío, se debe ignorar. El final del array se indica mediante la clave **CAL\_KV\_END**.

flags	Cada bit de este entero indica si un cierto tipo de transformación está presente o no. El significado de cada bit viene dado por las macros <b>CAL_FLAG_...</b>
-------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------

**afin\_1** es lo que Aerotri llama una transformación previa. Típicamente se usa para pasar del sistema píxel original de la foto, con el origen en la esquina superior izquierda y el eje *y* hacia abajo, a un sistema con origen en el centro de la foto y el eje *y* hacia arriba, ya sea en píxeles o en milímetros. La manera más rápida de explicar **afin\_1** y el significado de algunos bits de **flags** es mostrando el código de la función *medida\_a\_fotocoord* de la librería:

```
void medida_a_fotocoord(Puntoxy_Double *ob,
                        const OrientacionInterna *interna){
    if(interna->flags & CAL_FLAG_Transf1){
        Double aux;
        ob->x-= interna->afin_1.Tx;
        ob->y-= interna->afin_1.Ty;
        aux= interna->afin_1.a*ob->x + interna->afin_1.b*ob->y;
        ob->y= interna->afin_1.c*ob->x + interna->afin_1.d*ob->y;
        ob->x= aux;
```



```

    }
    if(interna->flags & CAL_FLAG_pp){
        ob->x-= interna->valI.x;
        ob->y-= interna->valI.y;
    }
    if(interna->flags & CAL_FLAG_Distorsiones){
        corrige_distorsion(ob,interna,0);
    }
}

```

A su vez, para que se calcule cada tipo de distorsión: radial simétrica, etc. es necesario que el correspondiente bit en flags esté a 1.

valI	Almacena la focal y el punto principal, cuyo significado exacto se deriva del código arriba mostrado.
distorsion	Contiene un elemento de tipo <b>ConfigDistorsion</b> cuya exposición se desarrolla al explicar la estructura <b>Trabajo_Autocalib</b> . Contiene también cuatro arrays de float que almacenan los valores de las componentes de las cuatro series: Radial simétrica, Tangencial simétrica, Asimétricas 1 y Asimétricas 2.

El manual Calibracion.html que viene con el instalador de Aerotri explica detalladamente el modelo de distorsión de Aerotri.

## InternaKPB

```

typedef struct{
    uint8_t inexacta;
    struct{
        Double cx, cy; //center, in pixels with respect to top left corner
        float px, py;
    } pixels;
    Double x,y;
    Double f;
    float k[5]; //k[0]: distorsión lineal
    float p1,p2; //Half of these could go away
    float b1,b2; //The appearance of b1 changes k[0]; b2 would change kappa
} InternaKPB;

```

Esta estructura almacena una interna en la que los parámetros de distorsión son los conocidos K1, K2, K3, P1 y P2. Además permite un K4 y un K0, representando este último una distorsión radial lineal, proporcional al radio. El comentario en las componentes p1, p2

da una idea de porqué este modelo se llamaba antes InternaMala, si bien es de largo es más usado.

<code>inexacta</code>	Flags que indican pérdidas en un paso de <b>OrientacionInterna</b> a <b>InternaKPB</b> o bien cambios que darían lugar a orientaciones externas distintas. Véase el fichero <a href="#">clases_distorsion.h</a> para una explicación detallada.
<code>pixels</code>	<code>cx, cy</code> son análogos a <code>Tx, Ty</code> de <code>afin_1</code> en la interna de <code>Aerotri</code> (véase el comentario). <code>px, py</code> son los tamaños de píxel en <i>x</i> e <i>y</i> . Normalmente son iguales. Pueden ser iguales a 1, en cuyo caso las unidades del resto de la estructura son píxeles.
<code>x, y</code>	Posición del punto principal respecto al centro ( <code>cx, cy</code> ). Análogos a <code>valI.x</code> y <code>valI.y</code> de <code>Aerotri</code> .
<code>f</code>	La focal
<code>k[5], p1, p2</code>	Las componentes de distorsión. <code>k[0]</code> se establece inicialmente a cero al crear esta estructura a partir de una <b>OrientacionInterna</b> .
<code>b1, b2;</code>	Otras componentes. La aparición de <code>b1</code> cambia <code>k[0]</code> . Por tanto <code>k[0]</code> y <code>b1</code> son ambos cero o ambos distintos de cero si se generan a partir de una <b>OrientacionInterna</b> mediante las funciones de esta librería; <code>b2</code> cambiaría kappa.

## Transformación entre OrientacionInterna InternaKPB

La primera función transforma de KPB a Aerotri; la segunda en sentido contrario :

```
int interna_Aerotri___kpb(const InternaKPB *kpb,
                          OrientacionInterna *interna,
                          bint bfocal, bint bpp);

int interna_kpb___Aerotri(const OrientacionInterna *interna,
                          InternaKPB *kpb, float *despl);
```

Estos nombres un tanto extraños se deben a que en el editor de código del autor la secuencia de caracteres `___` se muestra gracia a una ligadura de la fuente como «, de modo que los nombres se ven como *interna\_Aerotri*«*kpb* y *interna\_kpb*«*Aerotri* respectivamente.

Véase el archivo de cabecera para el significado de `bfocal` y `bpp` y los valores recomendados así como para los posibles valores devueltos por ambas funciones. El puntero `despl` puede ser `NULL`. En caso contrario se almacena en `*despl` lo que ha sido necesario desplazar el punto principal al transformar a **InternaKPB**. Si es mayor que  $8 \cdot 10^{-6}$  de la semidiagonal del fotograma se pone a 1 el bit bajo de `inexacta` en `kpb`.

## CombinaInternas

```
int combina_internas(const OrientacionInterna *internal1,
                    const OrientacionInterna *interna2,
                    OrientacionInterna *combinada);
```

Esta función recibe dos orientaciones internas y genera, en una orientación interna única, el resultado de aplicar primero una y después la otra. *internal1* e *interna2* apuntan a orientaciones internas. *combinada* apunta a una orientación interna vacía, que la función rellena y que posteriormente ha de liberarse mediante *free\_interna*. Se usa normalmente para combinar la interna procedente del fichero de cámara con la que surge del cálculo de parámetros de autocalibración.

## Modificaciones en la interna

Las funciones *elimina\_a1*, *elimina\_a1\_semidiag* y *cambia\_semidiag* modifican la orientación interna pero de manera que el resultado neto es el mismo. Esto es posible ya que los parámetros no son todos independientes, y variaciones de unos se pueden anular con variaciones de otros. Se emplean por otras funciones que manejan orientaciones internas: las que transforman una **OrientacionInterna** en una **InternaKPB** o viceversa y *combina\_internas*. Si el programador entiende lo que hacen (es imposible explicarlo brevemente con precisión) puede emplearlas si las necesita.