GammaLib SDD

Software Design Description

Version draft
23 May 2010

Author: Jürgen Knödlseder
Approved by: Jürgen Knödlseder

Institut de Recherche en Astrophysique et Planétologie (IRAP)
9, avenue du Colonel-Roche
31028 Toulouse Cedex 4
FRANCE

This page intentionally left blank

Contents

1 Introduction

1.1 Design Overview

1.2 Requirements Traceability Matrix

2 System Architectural Design

2.1 Chosen System Architecture

The `GammaLib` is designed for Linux, Unix and MacOS X systems and compiles under 32 Bit and 64 Bit.

2.2 Discussion of Alternative Designs

2.3 System Interface Description

The `GammaLib` is designed as C++ API library of which all modules are designed as classes. A Python interface allows scripting of library components.

3 Detailed Description of Components

3.1 Applications

3.1.1 `GApplication`

3.1.2 `GPars`

3.1.3 `GLog`

3.2 Skymaps

3.2.1 `GSkymap`

The `GSkymap` class holds a single skymap (or a set of skymaps with identical coordinates) in a specific world coordinate projection (see `GWcs`). Three different constructors exist:

```
GSkymap(void);
GSkymap(std::string wcs, std::string coordsys, int nside, std::string ordering,
        int nmaps = 1);
GSkymap(std::string wcs, std::string coordsys, GSkyDir dir, int nlon, int nlat,
        double dlon, double dlat, int nmaps = 1);
```

The first one creates an instance of an empty skymap.

The second is for creating an instance of a Healpix skymap. Here `wcs` needs to be `HPX` (we add this for consistency), `coordsys` is the coordinate system (see below), `nside` is the N_{side} parameter, `ordering` is the pixel ordering (either `RING` or `NESTED`, case independent), and `nmaps` gives the number of maps in a set (defaults to 1 if not given).

The third is for creating an instance of any other type of skymap. Here `wcs` is the projection of the skymap (see `GWcs`), `coordsys` is the coordinate system (see below), `dir` is the position of the map centre, `nlon` is the number of pixels in longitude, `nlat` is the number of pixels in latitude, `dlon` is the pixel size in longitude (in units of degrees), `dlat` is the pixel size in latitude (in units of degrees), and `nmaps` gives the number of maps in a set (defaults to 1 if not given).

Two coordinate systems are implemented (`coordsys`): `GAL` indicates the galactic system (galactic longitude and latitude), and `CEL` indicates equatorial or celestial system (Right Ascension and declination).

The following methods are defined for `GSkymap`:

```
void        read(std::string filename);     //!< Read skymap from FITS file
void        read(const GFitsHDU* hdu);      //!< Read skymap from FITS HDU
void        write(std::string filename);    //!< Write skymap into FITS file
void        write(GFitsHDU* hdu);           //!< Write skymap into FITS HDU
std::string ordering(void) const;           //!< Returns Healpix ordering ("" if not Healpix)
std::string coordsys(void) const;           //!< Returns coordinate system
GSkyDir     pix2dir(const int& pix);        //!< Convert pixel index to sky direction
int         dir2pix(GSkyDir dir) const;     //!< Convert sky direction to pixel index
double      omega(const int& pix);          //!< Return solid angle of pixel (in steradian)
int         npix(void) const;               //!< Return number of pixels in image
int         nside(void) const;              //!< Return number of sides (0 if not Healpix)
int         nlon(void) const;               //!< Return number of longitude pixels (0 if Healpix)
int         nlat(void) const;               //!< Return number of latitude pixels (0 if Healpix)
void        ordering(std::string ordering); //!< Change Healpix ordering (RING or NESTED)
void        coordsys(std::string coordsys); //!< Change coordinate system (CEL or GAL)
void        nside(int nside);               //!< Change Healpix N_side parameter
void        rebin(int nlon, int nlat);      //!< Change number of longitude/latitude pixels
void        smooth_gaussian(double sigma);  //!< Smooth skymap using Gaussian kernel
```

The `GSkymap` class contains:

```
GWcs*   m_wcs;     //!< Pointer to World Coordinate System projection
double* m_pixels;  //!< Pointer to sky map pixels
```

Note that the skymap is internally stored in a double precision array. The class may eventually by extended to also handle other storage types. Alternatively, the class could also be implemented as a template class.

3.2.2 `GWcs`

Virtual base class that implements projections from sky coordinates into a vector of image pixels. The class follows the World Coordinate System (WCS) convention.

The following pure virtual methods are defined for `GWcs`:

```
void    read(const GFitsHDU* hdu);     //!< Read projection information from FITS header
void    write(GFitsHDU* hdu);          //!< Write projection information to FITS header
GSkyDir pix2dir(const int& pix);       //!< Convert pixel index to sky direction
int     dir2pix(GSkyDir dir) const;    //!< Convert sky direction to pixel index
double  omega(const int& pix);         //!< Return solid angle of pixel (in steradian)
int     npix(void) const;              //!< Return number of pixels in image
int     naxes(void) const;             //!< Return number of image axes
int     naxis(const int& axis) const;  //!< Return number of pixels in specified axis
```

The HEALPix projection (as a general class of spherical projections) is represented by the keyword HPX in the FITS standard for writing astronomical data files.

3.3 Observation handling

3.3.1 `GObservation`

Observation handling provides an abstract interface to gamma-ray observations of all kinds without any reference to instrument specific properties. An observation is defined as a given period in time during which data are acquired with a specific instrument in a specific configuration. The data taking period needs not to be continuous. Good time intervals will define continous data taking periods within a given observation.

The basic element of observation handling is the `GObservation` abstract base class. It contains:

```
GEvents*    m_events;       //!< Pointer to events
GGti*       m_gti;          //!< Pointer to good time intervals
GResponse*  m_response;     //!< Pointer to instrument response function
```

3.3.2 `GObservations`

Observations taken with different instruments or with different instrument configurations are gathered together in a `GObservations` container class which provides an abstract interface to all data. **This is the main user interface class.**

The following methods are defined for `GObservations`:

```
void     append(GObservation &obs);      //!< Append observation
int      size(void) const;               //!< Returns number of observations
void     models(const GModels& models);  //!< Set models
GModels* models(void);                   //!< Returns pointer to models
void     optimize(GOptimizer& opt);      //!< Optimize model parameters
```

Events can be accessed via an iterator in the following way

```
GObservations obs;
...
for (GObservations::iterator event = data.begin(); event != data.end(); ++event) {
    cout << *event << endl;
}
```

3.3.3 `GEnergy`

Energy units used in data may differ between instruments. For this reason, energies are handled by a unified `GEnergy` class that internally stores energy information in MeV, and that provides methods for automatic conversion of energies. Mathematical operations may be performed on `GEnergy` as if it were a double precision variable.

3.3.4 `GTime`

Also time systems used in data may differ between instruments. For this reason, times are handled by a unified `GTime` class that internally stores time in MJD, and that provides methods for automatic conversion

between the various time systems that are used. Mathematical operations may be performed on `GTime` as if it were a double precision variable.

3.4 Event handling

3.4.1 `GEvent`

The fundamental unit of gamma-ray data are events. Events are realized in `GammaLib` by the abstract `GEvent` class. Two fundamental event types exists which derive from `GEvent`: event atoms, i.e. individual events which are realized by the abstract `GEventAtom` class, and event bins, i.e. bins of an event cube which are realized by the abstract `GEventBin` class. Derived from these two classes are the instrument specific event classes `GXXXEventAtom` and `GXXXEventBin` (where `XXX` is the instrument code) which implement the instrument specific methods. The class dependencies are shown in Fig. ??.

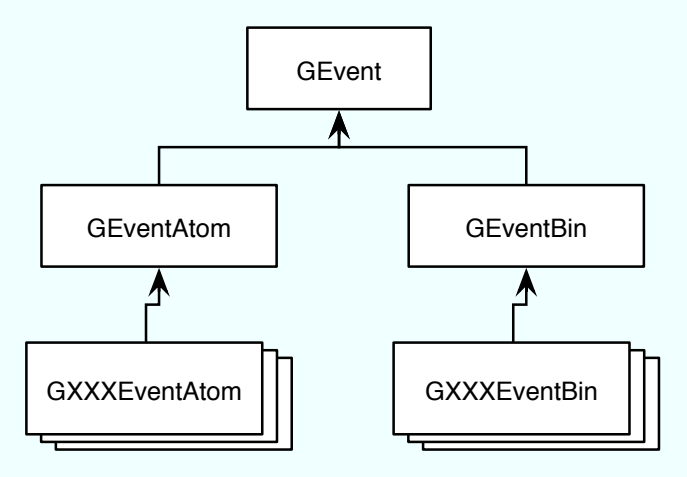

Figure 1: Class dependencies for the `GEvent` class.

The following virtual methods are defined for `GEvent`:

```
double   counts(void) const;                          //!< Number of counts in event
double   model(GModels& models);                      //!< Model value
double   model(GModels& models, GVector* gradient);   //!< Model value and gradient
GTime*   time(void);                                  //!< Time tag of event
GEnergy* energy(void);                                //!< Event energy
GSkyDir* dir(void);                                   //!< Arrival direction of event
bool     isatom(void) const;                          //!< Event is an atom
bool     isbin(void) const;                           //!< Event is a bin
```

3.4.2 `GEvents`

The basic element of event handling is the `GEvents` abstract container base class that contains `GEvent` objects. The derived abstract container class `GEventList` holds `GEventAtom` objects, while the derived abstract container class `GEventCube` holds `GEventBin` objects. Derived from these two classes are the instrument specific container classes `GXXXEventList` and `GXXXEventCube` (where `XXX` is the instrument code) which implement the instrument specific methods. The class dependencies are shown in Fig. ??. The following generic methods are implemented:

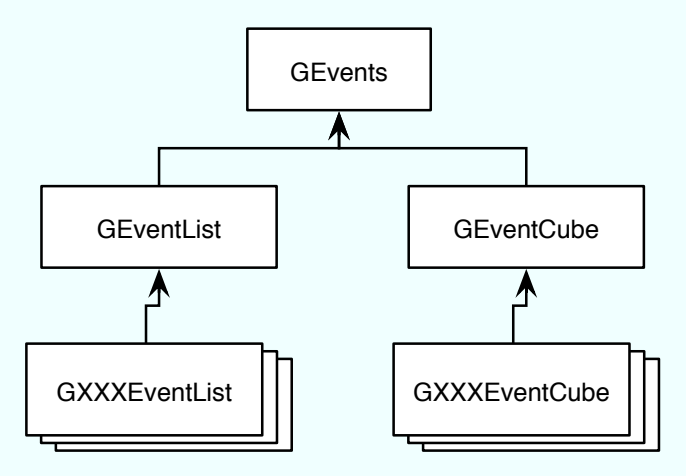

Figure 2: Class dependencies for the `GEvents` class.

```
void    load(const std::string& filename);   //!< Load events from FITS file
GEvent* pointer(int index) const;            //!< Returns pointer to event (atom or bin)
int     number(void) const;                  //!< Returns total number of events
int     elements(void) const;                //!< Returns total number of elements
```

Note that for an event list the total number of elements is identical to the total number of events.

`GEvents` allows for iterating to provide successive access to events. The example below shows how iterating is implemented (illustrated using the `GLATEvents` derived class):

```
GLATEvents events;
...
for (GLATEvents::iterator event = events.begin(); event != events.end(); ++event) {
    cout << *event << endl;
}
```

3.5 Model handling

`GModel` desribes how the gamma-ray (or background event) intensity I depends on the direction $\vec{d}$ on the sky, on the photon energy E_γ, and on the time t as function of a number of parameters $\vec{p}$. It is assumed that the spatial, spectral and temporal description can be factorized following

$$I(\vec{d}, E_\gamma, t|\vec{p}) = I(\vec{d}|\vec{p}) \times I(E_\gamma|\vec{p}) \times I(t|\vec{p}) \quad (1)$$

3.5.1 `GModelPar`

The elements of the parameter vector $\vec{p}$ are realized by the `GModelPar` class

```
class GModelPar {
...
protected:
  std::string m_name;      //!< Parameter name
  std::string m_unit;      //!< Parameter unit
  double      m_value;     //!< Parameter value
```

```
  double      m_error;     //!< Uncertainty in parameter value
  double      m_min;       //!< Parameter value minimum
  double      m_max;       //!< Parameter value maximum
  double      m_scale;     //!< Parameter scaling (real = m_value * m_scale)
  bool        m_free;      //!< Parameter is free/fixed (for fitting)
  bool        m_hasmin;    //!< Parameter has minimum boundary
  bool        m_hasmax;    //!< Parameter has maximum boundary
}
```

3.5.2 GModel

The model is realized by the `GModel` class

```
class GModel {
...
  std::string name(void) const;               //!< Return model name
  void        name(const std::string& name);  //!< Set model name
  int         npars(void) const;              //!< Returns number of parameters
  GModelPar*  par(int) const;                 //!< Returns pointer to parameter
protected:
  int             m_npars;        //!< Total number of parameters in model
  GModelPar**     m_par;          //!< Pointers to all model parameters
  GModelSpatial  *m_spatial;      //!< I(d|p)
  GModelSpectral *m_spectral;     //!< I(E|p)
  GModelTemporal *m_temporal;     //!< I(t|p)
}
```

3.5.3 GModelSpatial

The abstract base class `GModelSpatial` implements different functional forms for $I(\vec{d}|\vec{p})$. The derived classes contain the model parameters, and `m_par` provides pointers to all these parameters for quick access. As an example, the `GModelSpatialPtsrc` derived class is show:

```
class GModelSpatialPtsrc {
...
  int        npars(void) const;   //!< Returns number of parameters
  GModelPar* par(int) const;      //!< Returns pointer to parameter
protected:
  GModelPar  m_ra;                //!< Right Ascension of source
  GModelPar  m_dec;               //!< Declination of source
}
```

The following classes are implemented:

```
GModelSpatialPtsrc      //!< Point source
```

3.5.4 GModelSpectral

The abstract base class `GModelSpectral` implements different functional forms for $I(E_\gamma|\vec{p})$. The following classes are implemented:

```
GModelSpatialPlaw       //!< Power law
```

3.5.5 GModelTemporal

The abstract base class `GModelTemporal` implements different functional forms for $I(t|\vec{p})$. The following classes are implemented:

```
GModelTemporalConst     //!< Constant source
```

3.5.6 GModels

The models are collected into the `GModels` container class

```
class GModels : public GOptimizerPars {
  void        add(GModel);        //!< Add model to container
  int         npars(void) const;  //!< Returns number of parameters
  GModelPar*  par(int) const;     //!< Returns pointer to parameter
...
  int         m_elements;         //!< Total number of models
  GModel*     m_model;            //!< Pointers to all models
  int         m_npars;            //!< Total number of model parameters
  GModelPar** m_par;              //!< Pointers to all model parameters
}
```

which is derived from the general parameter container class `GOptimizerPars`. It is this container class that is handed to `GData` in order to compute the model prediction for a some given data. The interpretation of the models is done on the level of the instrument specific classes `GXXXEventList` and `GXXXEventCube` (where `XXX` is the instrument code).

`GModels` is also handed to the `GOptimizer` class for parameter optimization.

The following code illustrates how a point source is set up for the Crab pulsar with a power law spectral shape:

```
// Define Crab position
GSkyDir dir;
dir.radec_deg(83.6331, +22.0145);

// Spatial model is a point source at the position of the Crab
GModelSpatialPtsrc point_source = GModelSpatialPtsrc(dir);

// Spectral model is a power law
GModelSpectralPlaw power_law = GModelSpectralPlaw(1.0e-7, -2.1);

// Set up Crab model
GModel crab = GModel(point_source, power_law);

// Create model container
GModels models;
models.add(crab);

// Set model for observation
GData data;
...
data.set_model(models);
```

3.6 Optimizer

3.6.1 `GOptimizerFunction`

The abstract base class `GOptimizerFunction` provides the interface between any user-defined function and the `GOptimizer` class. The `GOptimizerFunction` class has the following structure:

```
class GOptimizerFunction {
  ...
  virtual void           eval(const GOptimizerPars& pars); //!< Evaluate function
  virtual double*        value(void);        //!< Return pointer to function value
  virtual GVector*       gradient(void);     //!< Return pointer to gradient
  virtual GSparseMatrix* covar(void);        //!< Return pointer to covariance matrix
}
```

3.6.2 `GOptimizer`

The `GOptimizer` class ...

```
class GOptimizer {
  ...
  virtual GOptimizerPars& operator() (GOptimizerFunction& fct, GOptimizerPars& p);
  GModels& operator() (GOptimizerFunction& fct, GModels& m);
  }
}
```

Optimization is then done using

```
void GData::optimize(GOptimizer& opt)
{
  // Set optimizer function
  GData::optimizer fct(this);

  // Optimise model parameters
  m_models = opt(fct, m_models);

  // Return
  return;
}
```

4 User Interface Design

4.1 Description of the User Interface

5 Additional Material