A Likelihood Tool Prototype for Analyzing GLAST/LAT Data

Jim Chiang
GLAST Science Support Center
Pat Nolan, Karl Young, Toby Burnett
LAT Team

Preliminaries

- **Instrument Response Functions**: It is conventional to represent the instrument response in terms of three functions:

$$D(E'; E, \hat{p}) \equiv \text{Energy Dispersion} \tag{1}$$

$$P(\hat{p}'; E, \hat{p}) \equiv \text{Point Spread Function} \tag{2}$$

$$A(E, \hat{p}) \equiv \text{Effective Area} \tag{3}$$

The energy dispersion and point spread function are the probability densities of measuring an apparent energy E' and apparent direction $\hat{p}'$, respectively, for a detected photon that has true energy E and direction $\hat{p}$.

The total response of the instrument is given by

$$R(E', \hat{p}'; E, \hat{p}) = D(E'; E, \hat{p}) P(\hat{p}'; E, \hat{p}) A(E, \hat{p}) \tag{4}$$

$$= \frac{d\sigma}{dE' d\hat{p}'}(E, \hat{p}). \tag{5}$$

As the latter relation indicates, this quantity can be interpreted as the differential cross-section of the telescope. Note that the total response need not factor into the three constituent parts, D, P, and A.

- **Gamma-Ray Source Model**: The photon specific intensity from a source i will be denoted by

$$\left.\frac{dN}{dE d\hat{p}\, dA dt}\right|_i = S_i(E, \hat{p}, t). \tag{6}$$

For point sources, such as pulsars or AGNs, the angular distribution of photons on the sky is a δ-function:

$$S_i(E, \hat{p}, t) = s_i(E, t)\delta(\hat{p} - \hat{p}_i). \tag{7}$$

Diffuse sources include the Galaxy and the extragalactic diffuse emission (the latter of which may actually comprise unresolved AGNs), as well as discrete extended sources such as a supernova remnant or the LMC.

- Given the above definitions, the expected distribution of detected photons is

$$
\begin{aligned}
M(E', \hat{p}', t) &= \int dE d\hat{p} R(E', \hat{p}'; E, \hat{p}, t) S(E, \hat{p}, t) \qquad (8) \\
&= \sum_i M_i(E', \hat{p}', t) \qquad (9)
\end{aligned}
$$

where

$$S(E, \hat{p}, t) \equiv \sum_i S_i(E, \hat{p}, t), \qquad (10)$$

and

$$M_i(E', \hat{p}', t) = \int dE d\hat{p} R(E', \hat{p}'; E, \hat{p}, t) S_i(E, \hat{p}, t). \qquad (11)$$

A time-dependence has been added to the total response to account for the varying orientation of the LAT with respect to the Celestial sphere as the instrument scans. The integrals in these expressions should in principle be evaluated over all possible true energies, $0 < E < \infty$, and over all possible directions, $\hat{p} \in 4\pi\,\mathrm{sr}$. In practice, the limited range of the LAT response functions will allow us to impose cut-offs at finite energies and over smaller solid angles.

- **Unbinned log-Likelihood**: Labeling individual photon events with the index j, the logarithm of the (unbinned) Poisson likelihood is

$$\log \mathcal{L} = \sum_j \left[\log M(E'_j, \hat{p}'_j, t_j)\right] - \sum_i N_{i,\mathrm{pred}} \tag{12}$$

where the predicted number of photons from source i is

$$N_{i,\mathrm{pred}} = \int_{\mathrm{ROI}} dE' d\hat{p}' dt M_i(E', \hat{p}', t) \tag{13}$$

The $N_{i,\mathrm{pred}}$ integrals are performed over the extraction region or "region-of-interest" (ROI), which is the volume of the (E', $\hat{p}'$, t) data space that is being considered.

We shall refer to the first term of eq. 12 as the "data sum" and the second term as the "model integral".

`Function`, `Arg`, `Parameter` Classes

In order to translate the preceding description to `C++`, we use these classes for the implementation of the various constituents of the likelihood calculation.

A Function object implements

$$F(x; \vec{\alpha}), \tag{14}$$

where

- F ← `Function` object.
- x ← `Arg` object. These guys are passed by reference to the `Arg` abstract base class, with the concrete sub-classes wrapping various kinds of data — `dArg` ⇔ `double`, `EventArg` ⇔ `Event`, `SkyDirArg` ⇔ `astro::SkyDir`, etc..
- $\vec{\alpha}$ ← vector of `Parameter` objects, each of which encapsulates the information we wish to associate with each model parameter: value, upper and lower bounds, scale factor, whether its free or fixed, etc..

Parameter Class

Each `Function` object has a `vector` of these `Parameter` objects as a data member called `m_parameter`. Many of the methods provided by the `Function` class allow for the `Parameter`s and derivatives of the `Function` object with respect to those `Parameter`s to be accessed singly or in groups. The group access methods are particularly important as they provide the means by which the `Optimizer` objects interact with the `Statistic` objects:

```
void Function::setFreeParamValues(const std::vector<double> &paramVec) {
   if (paramVec.size() != getNumFreeParams()) {
// ...exception stuff...
   } else {
      std::vector<double>::const_iterator it = paramVec.begin();
      setFreeParamValues_(it);
   }
}
std::vector<double>::const_iterator Function::setFreeParamValues_(
   std::vector<double>::const_iterator it) {
   for (unsigned int i = 0; i < m_parameter.size(); i++)
      if (m_parameter[i].isFree()) m_parameter[i].setValue(*it++);
   return it;
}
```

The method `setFreeParamValues_(...)` is reimplemented in `CompositeFunction` and `SourceModel`.

`Arg` Class

By wrapping an argument x in an `Arg` sub-class, one can have the resulting object passed by reference in the argument lists of `Function`'s methods . This allows the derivative access methods of `Function` to be inherited by sub-classes transparently even though the underlying data-types of the `Function` arguments differ. Access to the data in the concrete sub-classes (`dArg`, `EventArg`, `SkyDirArg`, `SrcArg`) is achieved in the `Function` sub-class implementations by down-casting. For example, the following implements eq. 12:

```
double logLike_ptsrc::value(const std::vector<double> &paramVec) {
   setFreeParamValues(paramVec);   // inherited from Function via
                                   // SourceModel's setFreeParamValues_ method
   double my_value = 0;
// the "data sum"
   for (unsigned int j = 0; j < m_events.size(); j++) {
      EventArg eArg(m_events[j]);
      my_value += m_logSrcModel(eArg); // m_logSrcModel's parameters are updated
                                       // automatically through SourcModel::s_sources
   }
// the "model integral", a sum over Npred for each source
   for (unsigned int i = 0; i < getNumSrcs(); i++) {
      SrcArg sArg(s_sources[i]);
      my_value -= m_Npred(sArg);
   }
   return my_value;
}
```

The real utility of `Arg`:

```
void Function::fetchDerivs(Arg &x, std::vector<double> &derivs, bool getFree) const {
   if (!derivs.empty()) derivs.clear();
   for (unsigned int i = 0; i < m_parameter.size(); i++) {
      if (!getFree || m_parameter[i].isFree())
         derivs.push_back(derivByParam(x, m_parameter[i].getName()));
   }
}
double PowerLaw::derivByParam(Arg &xarg, const std::string &paramName) const {
   double x = dynamic_cast<dArg &>(xarg).getValue();
//... consistency checks, exceptions, etc....
   switch (iparam) {
   case Prefactor:
      return value(xarg)/my_params[Prefactor].getTrueValue()
         *my_params[Prefactor].getScale();
      break;
   case Index:
      return value(xarg)*log(x/my_params[Scale].getTrueValue())
         *my_params[Index].getScale();
      break;
//...
   default:
      break;
   }
   return 0;
}
```

`Function` Sub-Classes for Source Modeling

- `PowerLaw`, `ConstantValue`, `Gaussian`, `AbsEdge`: These are the building blocks of modeling sources. More such classes can be added by us, or by clients, if desired.
- `CompositeFunction` (`ProductFunction`, `SumFunction`): These classes can be used to combine existing `Function` objects that have the same `Arg`-type to produce more complicated `Function`s for modeling source characteristics without requiring the client to provide new `Function` sub-classes.
- `SkyDirFunction`: This class wraps a `SkyDir` object in a `Function` context so that sky coordinates, such as RA and Dec, can be treated as fit parameters.
- `SpatialMap`: This class allows a FITS image file to serve as a template for the spatial distribution of emission from an extended source. The Galactic diffuse model used by EGRET is an example.

Source Classes

- Gamma-ray sources must implement the following four methods (which are pure virtual functions in the `Source` base class):
  - `fluxDensity(Event &)` $= M_{ij} \equiv M_i(E'_j, \hat{p}'_j, t_j)$ for a source i and a photon event j.
  - `fluxDensityDeriv(Event &, string &paramName)` $= \partial M_{ij}/\partial\alpha$ for a parameter α.
  - `Npred()` $= N_{i,\mathrm{pred}} = \int_{\mathrm{ROI}} M_i(E', d\hat{p}', t) dE' d\hat{p}' dt$.
  - `NpredDeriv(string &paramName)` $= \partial N_{i,\mathrm{pred}}/\partial\alpha$.

  The first two methods are independent of the ROI cuts and do not pertain to any particular fit statistic. The latter two methods are used specifically for unbinned likelihood and are implemented differently for point-like and diffuse sources.

- Spatial and spectral components are assumed to factor, with separate `Function` objects describing each. From `PointSource`:

```
void setDir(const astro::SkyDir &dir, bool updateExposure = true) {
   m_dir = SkyDirFunction(dir);
   m_functions["Position"] = &m_dir;
   if (updateExposure) computeExposure();
}
void setSpectrum(Function *spectrum) {
   m_spectrum = spectrum->clone();
   m_functions["Spectrum"] = m_spectrum;
}
```

`PointSource` Class

Because their spatial distribution is a δ-function, point-like sources are relatively straight-forward to implement. For a source i with fixed location, $\hat{p}_i$, one need only integrate the exposure at that location once at the outset of the calculation,

$$\varepsilon_i(E) = \int_{\rm ROI} dE' d\hat{p}' dt R(E', \hat{p}'; E, \hat{p}_i, t), \tag{15}$$

so that the predicted number of photons for this source is given by

$$N_{i,\rm pred} = \int dE s_i(E; \vec{\alpha}_i) \varepsilon_i(E). \tag{16}$$

Important simplifications:

- The source spectrum is constant — usually ok for relatively small numbers of photons.
- The source location is not allowed to vary in the fitting process.
  - The boundary of the data space, i.e., the ROI, can be complex owing to zenith angle cuts (to limit Earth albedo contribution), etc..
  - PSF may not have many symmetries.
  - Fixed source location allows eq. 15 to be computed *once* for a given set of ROI cuts.

DiffuseSource Class

As we have mentioned, we assume, largely for tractability, that the photon specific intensity from a `DiffuseSource` object factors into separate spectral and spatial components:

$$S_i(E, \hat{p}) = s_i(E)\tilde{S}_i(\hat{p}) \tag{17}$$

Therefore, in order to have spectral variation across an extended source, that source must be composed of a sufficient number of smaller `DiffuseSource` objects, each having its own `Function` object to model its spectrum. This approach is consistent with tessellation schemes, such as HTM or HEALPix, that have been proposed to model the diffuse Galactic emission.

Because of the extended nature of $\tilde{S}_i(\hat{p})$, the implementation of `DiffuseSource` is assisted by the following two classes:

- `SpatialMap`:
  - Objects of this class return interpolated values from a FITS image file as a function of `SkyDir` position. This allows `DiffuseSource` objects to use a FITS image as a template for the spatial distribution of the source emission.
  - However, `DiffuseSource` objects can use any `Function` object that returns a scalar value as a function of `SkyDir` position, e.g., `ConstantValue` is used to model the extragalactic diffuse emission as isotropic.

- `ExposureMap`:
  - This is a frame stack (or data cube) of exposure as a function of true energy and sky position.
  - A map can be computed, if desired, by the `computeMap(...)` method, which creates an array of `PointSource` objects and uses the `PointSource::computeExposure(...)` method (eq. 15). NB: For fitting of `PointSource` locations, exposures could be interpolated from an appropriate `ExposureMap` object.

For each `DiffuseSource` object i, the spatially-integrated exposure is calculated as a function of energy using the exposure map, which we denote by $\varepsilon(E, \hat{p})$:

$$\varepsilon_i(E) = \int d\hat{p}\tilde{S}_i(\hat{p})\varepsilon(E, \hat{p}). \tag{18}$$

The method `DiffuseSource::integrateSpatialDist()` computes this integral by calling the `ExposureMap::integrateSpatialDist(...)` method. Armed with the resulting $\varepsilon_i(E)$, eq. 16 then gives the predicted number of photons for `DiffuseSource` objects, just as it does for `PointSource` objects.

`SourceModel` Classes

- Objects of this class are composites of `Source` objects, which are in turn composites of `Function` objects. The `Source` objects are stored as cloned pointers in the static data member vector `s_sources`. This ensures that the sub-classes, `Statistic` and `logSrcModel`, use the same set of `Source` objects and `Parameter`s in their calculations.

```
void SourceModel::addSource(Source *src) {
// loop over sources to ensure unique names
   for (unsigned int i = 0; i < s_sources.size(); i++)
      assert((*src).getName() != (*s_sources[i]).getName());
// add a clone of this Source to the vector
   s_sources.push_back(src->clone());
// add the Parameters to the m_parameter vector
   Source::FuncMap srcFuncs = (*src).getSrcFuncs();
   Source::FuncMap::iterator func_it = srcFuncs.begin();
   for (; func_it != srcFuncs.end(); func_it++) {
      std::vector<Parameter> params;
      (*func_it).second->getParams(params);
      for (unsigned int ip = 0; ip < params.size(); ip++)
         m_parameter.push_back(params[ip]);
   }
}
```

- Group parameter access methods are based on Hippodraw's `FunctionBase` class. Iterators for the parameter value `vector` are passed and returned that allow the `set[Free]ParamValues_()` methods to be called in succession:

```
std::vector<double>::const_iterator SourceModel::setFreeParamValues_(
   std::vector<double>::const_iterator it) {
   for (unsigned int i = 0; i < s_sources.size(); i++) {
      Source::FuncMap srcFuncs = (*s_sources[i]).getSrcFuncs();
      Source::FuncMap::iterator func_it = srcFuncs.begin();
      for (; func_it != srcFuncs.end(); func_it++)
         it = (*func_it).second->setFreeParamValues_(it);
   }
   syncParams();  // this updates m_parameter
   return it;
}
```

  Recall from `Function`:

```
std::vector<double>::const_iterator Function::setFreeParamValues_(
   std::vector<double>::const_iterator it) {
   for (unsigned int i = 0; i < m_parameter.size(); i++)
      if (m_parameter[i].isFree()) m_parameter[i].setValue(*it++);
   return it;
}
```

Response Classes: `Aeff`, `Psf`

- These classes are Singleton since only one set of instrument response data and one set of spacecraft data will be used to analyze any given dataset of photons.
- Instances of these classes are functors, but they do not inherit from `Function` since they do not have parameters that are adjusted in the fitting process. Also, there is no need to wrap their arguments using sub-classes of `Arg`. Their function call operators, `()`, are overloaded so that their return values can be accessed either as a function of instrument or sky coordinates.
- Example of use (see eq. 11 and note the effects of the δ-functions in sky location and energy):

```
double PointSource::fluxDensity(double energy, double time,
                                const astro::SkyDir &dir) const {
// Scale the energy spectrum by the psf value and the effective area
// and convolve with the energy dispersion (now a delta-function in
// energy), all of which are functions of time and spacecraft attitude
// and orbital position.
   Psf *psf = Psf::instance();
   Aeff *aeff = Aeff::instance();
   dArg energy_arg(energy);
   double spectrum = (*m_spectrum)(energy_arg);
   double psf_val = (*psf)(dir, energy, m_dir.getDir(), time);
   double aeff_val = (*aeff)(energy, m_dir.getDir(), time);
   return spectrum*psf_val*aeff_val;
}
```

The latResponse Package

A set of abstract base classes that are intended to define a minimal, but complete interface to the response functions:

```
class IPsf {
public:
   virtual ~IPsf() {}

   /// Pure virtual method to define the interface for the member
   /// function returning the point-spread function value.
   /// @param appDir Apparent (reconstructed) photon direction.
   /// @param energy True photon energy in MeV.
   /// @param srcDir True photon direction.
   /// @param scZAxis Spacecraft z-axis.
   /// @param scXAxis Spacecraft x-axis.
   virtual double value(const astro::SkyDir &appDir,
                        double energy,
                        const astro::SkyDir &srcDir,
                        const astro::SkyDir &scZAxis,
                        const astro::SkyDir &scXAxis) const = 0;
// other stuff....
};
```

"Choice" of response functions is determined at object creation. Consider the effective area constructors for GLAST25 vs EGRET:

```
class AeffGlast25 : public IAeff, public Glast25 {
public:
   AeffGlast25(const std::string &filename, int hdu)
      : Glast25(filename, hdu) {readAeffData();}
// other stuff...
};


class AeffEgret : public latResponse::IAeff, public RespEgret {
public:
   AeffEgret(int caltbl, int eclass, int tascco, int ivp, int tdmode = 0x0fff)
      : RespEgret(tdmode) {m_aeff.init(caltbl, eclass, tascco, ivp);}
// other stuff...
};
```

Both of these classes must implement the `IAeff::value(...)` method using the same interface.

Classes for Manipulating Data

- `Table`: For accessing FITS binary table files.
- `FitsImage`: For accessing FITS image files.
- `Event`: An n-tuple containing photon arrival time, apparent direction and energy, etc.. Also stored in this object is event-specific response information that is used in the calculation of `DiffuseSource::fluxDensity(Event &)`. In general, this information takes the form of an energy-dependent response:

$$r_{ji}(E) = \int d\hat{p}\tilde{S}_i(\hat{p})R({E'}_j, \hat{p}'_j; E, \hat{p}, t_j), \tag{19}$$

where $\tilde{S}_i(\hat{p})$ is the angular distribution of photons from source i. The flux density is then

$$M_i({E'}_j, \hat{p}'_j, t_j) = \int dE \, r_{ji}(E) s_i(E), \tag{20}$$

where $s_i(E)$ is the source spectrum. The r_{ji}s are computed using the `Event::computeResponse(...)` methods and are stored in the `Event::diffuse_response` data member vector.

- `RoiCuts`: Cuts in energy, direction, and time cuts used selecting of the photons for analysis.
- `ScData`: Spacecraft data including the instrument attitude, whether it's in the SAA, etc., all as a function of time.

Classes Used for Analysis

- `Statistic`: As part of the `SourceModel` hierarchy, these objects use the event and spacecraft data along with the source model to provide objective functions for fitting the model parameters. The unbinned log-likelihood is the canonical example, but any sort of `Statistic` object may be defined.
- `Optimizer`: The sub-classes of `Optimizer` implement various methods for finding the maxima of `Statistic` functions. Currently implemented are wrappers for the Minuit variable-metric method and the BFGS quasi-Newton method. Both methods can handle simply-bounded parameters.
- `Mcmc`: This class uses the variable-at-a-time, Metropolis-Hastings update method to sample parameter space and thereby characterize the posterior distribution embodied by a given `Statistic`. Prior distributions can be applied to allow for a Bayesian interpretation of the parameter uncertainties and significances.

Unit Tests

- `FunctionTest`: This class provides a standard set of tests that should be used to provide *minimal* verification of a `Function` sub-class. It checks for consistent `Parameter` access, compares `Function` evaluations with known values for a user-supplied vector of `Function Arg`s, compares the derivatives provided by the `Function` with numerical estimates, and it tests value and derivative access for free and fixed parameters.

Using the A1 Classes: Analyzing Simulated LAT Data

From the test program:

```
void fit_DiffuseSource() {
// center the ROI on 3C 279
   double ra0 = 193.98;
   double dec0 = -5.82;
   RoiCuts::setCuts(ra0, dec0, 20.);

// root name for the observation data files
   std::string obs_root = "diffuse_test_5";

// read in the spacecraft data
   std::string sc_file = test_path + "Data/" + obs_root + "_sc_0000";
   int sc_hdu = 2;
   ScData::readData(sc_file, sc_hdu);

   std::string expfile = test_path + "Data/exp_" + obs_root + "_new.fits";

// compute a new exposure map for these data
//   ExposureMap::computeMap(expfile, 30., 60, 60, 10);

// must read in the exposure file prior to creating the SourceFactory
// object since it contains DiffuseSources
   ExposureMap::readExposureFile(expfile);

   SourceFactory srcFactory;
```

```
   DiffuseSource *ourGalaxy = dynamic_cast<DiffuseSource *>
      (srcFactory.makeSource("Milky Way"));
   DiffuseSource *extragalactic = dynamic_cast<DiffuseSource *>
      (srcFactory.makeSource("EG component"));

   Source *_3c279 = srcFactory.makeSource("PointSource");
   _3c279->setDir(ra0, dec0);
   _3c279->setName("3C 279");

// create the Statistic
   logLike_ptsrc logLike;

// add the Sources
   logLike.addSource(ourGalaxy);
   logLike.addSource(extragalactic);
   logLike.addSource(_3c279);

// read in the data
   std::string event_file = test_path + "Data/" + obs_root + "_0000";
   logLike.getEvents(event_file, 2);

// There are a few options for computing the DiffuseSource Event responses:

// individually...
//     logLike.computeEventResponses(*ourGalaxy);
//     logLike.computeEventResponses(*extragalactic);

// by constructing a vector of the targeted DiffuseSources...
//     std::vector<DiffuseSource> srcs;
```

```
//     srcs.push_back(*ourGalaxy);
//     srcs.push_back(*extragalactic);
//     logLike.computeEventResponses(srcs);

// or the default way, for all of the DiffuseSources in the SourceModel...
   logLike.computeEventResponses();

// do the fit
   verbose = 3;
   Minuit myMinuitObj(logLike);
   myMinuitObj.find_min(verbose, .0001);

   std::vector<double> sig = myMinuitObj.getUncertainty();
   for (unsigned int i=0; i < sig.size(); i++) {
      std::cout << i << "  " << sig[i] << std::endl;
   }
   std::vector<std::string> srcNames;
   logLike.getSrcNames(srcNames);

// replace (or add) each Source in srcFactory for later use
   std::vector<std::string> factoryNames;
   srcFactory.fetchSrcNames(factoryNames);
   for (unsigned int i = 0; i < srcNames.size(); i++) {
      Source *src = logLike.getSource(srcNames[i]);
      srcFactory.replaceSource(src);
   }
} // fit_DiffuseSource
```

Using the `Source` Classes: the `SourceFactory` Constructor

```
SourceFactory::SourceFactory() {
// Add a PointSource modeled by a PowerLaw as the default

// Note that the default constructor is used here, which means that
// exposure will not be computed.  A setDir(ra, dec, [true]) will
// cause the exposure to be computed and thus requires prior
// specification of the ROI cuts and spacecraft data.
   PointSource ptsrc;

// Add a nominal PowerLaw spectrum.  Note that one needs to reset the
// Parameters from the default and add sensible bounds.
   SpectrumFactory specFactory;
   Function *powerLaw = specFactory.makeFunction("PowerLaw");

// Use a nominal Parameter set for now with Prefactor = 10 (assuming a
// scaling of 1e-9, set below), Index = -2, and Scale = 100 (MeV).
// Set the bounds here as well.
   std::vector<Parameter> params;
   powerLaw->getParams(params);
   params[0].setValue(10);            // Prefactor
   params[0].setScale(1e-9);
   params[0].setBounds(1e-3, 1e3);
   params[1].setValue(-2);            // Index
   params[1].setBounds(-3.5, -1);
   params[2].setValue(100);           // Scale (this is fixed by default)
   powerLaw->setParams(params);
```

```
   ptsrc.setSpectrum(powerLaw);
   addSource("PointSource", &ptsrc, true);

// Add the map-based Galactic Diffuse Emission model;
// assume that the FITS file is available in a standard place...
   std::string galfile = "../src/test/Data/gas.cel";
   SpatialMap galacticModel(galfile);
   galacticModel.setParam("Prefactor", 1.1*pow(100., 1.1));

   try {
      DiffuseSource ourGalaxy(&galacticModel);
      ourGalaxy.setName("Milky Way");

// Provide ourGalaxy with a power-law spectrum.
      PowerLaw gal_pl(pow(100., -2.1), -2.1, 100.);
      gal_pl.setName("gal_pl");
      gal_pl.setParamScale("Prefactor", 1e-5);
      gal_pl.setParamTrueValue("Prefactor", pow(100., -2.1));
      gal_pl.setParamBounds("Prefactor", 1e-3, 1e3);
      gal_pl.setParamBounds("Index", -3.5, -1);

      ourGalaxy.setSpectrum(&gal_pl);

      addSource("Milky Way", &ourGalaxy, true);
   } catch (ParameterNotFound &eObj) {
      std::cerr << eObj.what() << std::endl;
      throw;
   } catch (LikelihoodException &likeException) {
      std::cerr << "Likelihood::SourceFactory: "
```

```
                 << "Cannot create DiffuseSource Milkyway.\n"
                 << likeException.what() << std::endl;
   }

// Add an extragalactic diffuse component.
   ConstantValue egNorm(1.);
   egNorm.setParam("Value", 1., false);   // fix to unity

   try {
      DiffuseSource extragalactic(&egNorm);
      extragalactic.setName("EG component");

      PowerLaw eg_pl(2.09e-3*pow(100., -2.1), -2.1, 100.);
      eg_pl.setName("eg_pl");
      eg_pl.setParamScale("Prefactor", 1e-7);
      eg_pl.setParamTrueValue("Prefactor", 2.09e-3*pow(100., -2.1));
      eg_pl.setParamBounds("Prefactor", 1e-5, 1e2);
      eg_pl.setParamBounds("Index", -3.5, -1);
      extragalactic.setSpectrum(&eg_pl);

      addSource("EG component", &extragalactic, true);
   } catch (ParameterNotFound &eObj) {
      std::cerr << eObj.what() << std::endl;
      throw;
   } catch (LikelihoodException &likeException) {
      std::cerr << "Likelihood::SourceFactory: "
                 << "Cannot create DiffuseSource EG component.\n"
                 << likeException.what() << std::endl;
   }
}
```

To Do

- Add energy dispersion.
- Generalize $N_{\rm pred}$ calculation to include zenith angle cuts and non-axisymmetric Psf's.
- Implement more realistic response function representations.
- Analyze EGRET data.
- Implement `Observation` class to contain everything associated with an observation: `RoiCuts`, the `Event` data, `ScData`, `ExposureMap`.
- Refactorings:
  - Make `Statistic` a true `Function` sub-class. Have it take an `Observation` object as a constructor argument.
  - Coordinate the design of the FITS-related classes, `Table`, `FitsImage`, etc., with those of other Science Tools.
  - Use Strategy pattern in `Optimizer` class.
  - Move `setDir(...)` method from `Source` to `PointSource`.
- Make Python extensions part of the CMT build system.
- Write proper unit tests for all classes.