{{lastupdated_at}} by {{lastupdated_by}}

Fermi/LAT interface

Pass 7 IRF

Pass 7 IRF have been implement in GammaLib version 00-06-00. Please check [[Validation of Pass 7 IRFs]] for more information.

Livetime cubes

The Fermi/LAT Science Tools implement the livetime cube information as map_tools::Exposure objects. map_tools::Exposure derives from SkyExposure which is defined as

```
typedef BasicExposure<SkyBinner, healpix::CosineBinner> SkyExposure;
```

The BasicExposure class is a template class which has the following access operator that is used to retrieve exposure values:

```
  template<class F>
     double operator()(const astro::SkyDir& dir, const F& fun)const
  {
     const C& binner = m_sky[dir];
     return binner(fun);
  }
```

Here C is the healpix::CosineBinner class. The class F is typically the effective area class. The binner(fun) operator is implemented in healpix/healpix/CosineBinner by

```
  template<class F>
  double operator()(const F& f)const
  {
     double sum=0;
     int n(0);
     for(const_iterator it=begin(); it!=end_costh(); ++it, ++n){
        sum += (*it)*f(costheta(it));
     }
     return sum;

  }
```

The energy does not appear explicitly in these code fragments as it is stored as a member of the effective area helper class:

```
 class Aeff : public AeffBase {
 public:
    Aeff(double energy, int evtType, const Observation & observation);
    virtual ~Aeff() {}
 protected:
    double m_energy;
    int m_evtType;
    const Observation & m_observation;
    virtual double value(double cosTheta, double phi=0) const;
 };
```

Exposure computation using efficiency factors

The Fermi/LAT Science Tools compute the exposure with trigger rate- and energy-dependent efficiency corrections in Likelihood/Likelihood/ExposureCube.h where the following template class is implemented:

```
  template<class T>
  double value(const astro::SkyDir & dir, const T & aeff, double energy) const {
     double factor1(1), factor2(0);
     if (m_efficiencyFactor) {
        double met((m_tstart + m_tstop)/2.);
        m_efficiencyFactor->getLivetimeFactors(energy, factor1, factor2, met);
     }
     double value1(value(dir, aeff));
     double value2(0);
     double exposure(factor1*value1);
     if (factor2 != 0) {
        value2 = value(dir, aeff, true);
        exposure += factor2*value2;
     }
     if (exposure < 0) {
        throw std::runtime_error("ExposureCube::value: exposure < 0");
     }
     return exposure;
  }
```

An approximation is made in the computation of the efficiency factors which are the same for the front and the back section (see irfs/latResponse/src/EfficiencyFactor.cxx):

```
void EfficiencyFactor::getLivetimeFactors(double energy, double & factor1, double & factor2, double met) const {
   (void)(met);
   if (!m_havePars) {
      factor1 = 1;
      factor2 = 0;
      return;
   }
   double logE(std::log10(energy));

   // Since we do not have access to the front and back effective
   // areas separately, just return the averages.  We would really
   // want to return the averages weighted by effective area (as a function
   // of energy and theta).
   factor1 = (m_p1_front(logE) + m_p1_back(logE))/2.;
   factor2 = (m_p0_front(logE) + m_p0_back(logE))/2.;
}
```

In GammaLib we do not make this approximation as the response is fully separated in front and back response functions. The efficiency factors are returned for front and back separately:

```
double GLATLtCube::operator()(const GSkyDir& dir, const GEnergy& energy, const GLATAeff& aeff) const
{
    // Compute exposure
    double exposure = m_exposure(dir, energy, aeff);

    // Optionally compute livetime factors for trigger rate- and
    // energy-dependent efficiency corrections
    if (aeff.has_efficiency()) {

        // Compute correction factors
        double f1 = aeff.efficiency_factor1(energy);
        double f2 = aeff.efficiency_factor2(energy);
```

```
      // Compute correction
      double correction = m_weighted_exposure(dir, energy, aeff);

      // Set exposure
      exposure = f1 * exposure + f2 * correction;

   } // endif: corrections requested

   // Return exposure
   return exposure;
}
```

Below an example for the efficiency factors for P7REP_SOURCE_V15 at 200 MeV:

Parameter	**Science Tools**	**GammaLib**
factor1::front	-1.06527	-1.05736
factor2::front	2.28282	2.27408
factor1::back	-1.06527	-1.07317
factor2::back	2.28282	2.29157

Effective area

The effective area is accessed in the Fermi/LAT Science Tools using the ExposureCube::Aeff::value method (recall that the energy is stored in the member m_energy):

```
double ExposureCube::Aeff::value(double cosTheta, double phi) const {
  double inclination = acos(cosTheta)*180./M_PI;
  double epoch((m_observation.expCube().tstart() + m_observation.expCube().tstop())/2.);
  std::map<unsigned int, irfInterface::Irfs *>::const_iterator respIt = m_observation.respFuncs().begin();
  for ( ; respIt != m_observation.respFuncs().end(); ++respIt) {
    if (respIt->second->irfID() == m_evtType) {
      irfInterface::IAeff * aeff = respIt->second->aeff();
      double aeff_val = aeff->value(m_energy, inclination, phi, epoch);
      return aeff_val;
    }
  }
  return 0;
}
```

The inclination is given in degrees.

The effective area is implemented by the irfs/latResponse/Aeff.cxx class. The value method is used to access the effective area values:

```
double Aeff::value(double energy, double theta, double phi, double time) const {
  (void)(time);
  double costheta(std::cos(theta*M_PI/180.));
  if (costheta < m_aeffTable.minCosTheta()) {
    return 0;
  }
  if (costheta > 0.99999) {
    costheta = 0.99999;  // blech.
  }
  bool interpolate;
  double logE(std::log10(energy));
  double aeff_value(m_aeffTable.value(logE, costheta, interpolate=true)*1e4);
  double phi_mod(phi_modulation(logE, costheta, phi, interpolate=false));
  return aeff_value*phi_mod;
```

```
}
```

Note the unnecessary conversion of inclination to costheta (why is not simply costheta passed in this interface?). Effective area values are determined by bilinear interpolation in logE and costheta. The code also multiplies-in a Phi modulation:

```
double Aeff::phi_modulation(double logE, double costheta, double phi, bool interpolate) const {
  double par[2];
  if (!m_phiDepPars || !m_usePhiDependence) {
    return 1.;
  }
  m_phiDepPars->getPars(logE, costheta, par, interpolate);
  return phi_modulation(par[0], par[1], phi);
}
double Aeff::phi_modulation(double par0, double par1, double phi) const {
  if (phi < 0) {
    phi += 360.;
  }
  double norm(1./(1. + par0/(1. + par1)));
  double phi_pv(std::fmod(phi*M_PI/180., M_PI) - M_PI/2.);
  double xx(2.*std::fabs(2./M_PI*std::fabs(phi_pv) - 0.5));
  return norm*(1. + par0*std::pow(xx, par1));
}
```

The equivalent GammaLib code is

```
double GLATAeff::operator()(const double& logE, const double& ctheta)
{
   // Get effective area value
   double aeff = (ctheta >= m_min_ctheta)
               ? m_aeff_bins.interpolate(logE, ctheta, m_aeff) : 0.0;

   // Make sure that effective area is not negative
   if (aeff < 0.0) {
      aeff = 0.0;
   }

   // Return effective area value
   return aeff;
}
```

A variant of this operator takes also a phi value, yet **the effective area phi dependence is so far not implemented in GammaLib**.

Here a number of effective area values (in cm2) determined using the ScienceTools and GammaLib for the P7REP_SOURCE_V15 response for 200 MeV as function of costheta (no phi dependence was used in the Fermi/LAT Science Tools):

logE	costheta	ScienceTools	GammaLib
2.301 03	0.999 844	2334.4	2426.55
2.301 03	0.998 594	2334.4	2417.21
2.301 03	0.996 09	2334.4	2398.55

	4		
2.301 03	0.992 344	2334.4	2370.56
2.301 03	0.987 344	2333.23	2333.23
2.301 03	0.981 094	2286.58	2286.58
2.301 03	0.418 594	119.825	119.825
2.301 03	0.379 844	60.7843	60.7843
2.301 03	0.339 844	26.3115	26.3115
2.301 03	0.298 594	8.99221	8.99221
2.301 03	0.256 094	2.41828	2.41828
2.301 03	0.212 344	0.571 005	0.566 958

For costheta > 0.992344 (about 7°) the effective area in the Fermi/LAT Science Tools saturates while for GammaLib it still increases (maybe the ScienceTools bi-linear interpolator does not extrapolate?). The Fermi/LAT Science Tools determine the minimum costheta angle from the lower boundary of the front response (note that formerly this was fixed to 70°, but I now dropped this since Fermi/LAT Science Tools do no limit the response).

Mean PSF

The Fermi/LAT Science Tools and GammaLib use a mean PSF for point sources in binned analysis. The mean PSF is implemented by Likelihood/src/MeanPsf.cxx:

```
void MeanPsf::init() {
  computeExposure();
  if (s_separations.size() == 0) {
    createLogArray(1e-4, 70., 200, s_separations);
  }
  m_psfValues.reserve(m_energies.size()*s_separations.size());
  for (unsigned int k = 0; k < m_energies.size(); k++) {
    for (unsigned int j = 0; j < s_separations.size(); j++) {
      double value(0);
      std::map<unsigned int, irfInterface::Irfs *>::const_iterator
        resp = m_observation.respFuncs().begin();
      for (; resp != m_observation.respFuncs().end(); ++resp) {
```

```
          int evtType = resp->second->irfID();
          Psf psf(s_separations[j], m_energies[k], evtType, m_observation);
          value += m_observation.expCube().value(m_srcDir, psf,
                                                 m_energies[k]);
        }
        if (m_exposure[k] > 0) {
          value /= m_exposure[k];
        } else {
          value = 0;
        }
        m_psfValues.push_back(value);
      }
    }
  }
  void MeanPsf::computeExposure() {
    m_exposure.reserve(m_energies.size());
    for (size_t k(0); k < m_energies.size(); k++) {
      double value(0);
      std::map<unsigned int, irfInterface::Irfs *>::const_iterator
        resp = m_observation.respFuncs().begin();
      for (; resp != m_observation.respFuncs().end(); ++resp) {
        int evtType = resp->second->irfID();
        ExposureCube::Aeff aeff(m_energies[k], evtType, m_observation);
        value += m_observation.expCube().value(m_srcDir, aeff, m_energies[k]);
      }
      m_exposure.push_back(value);
    }
  }
```