Adding Dither Corrections to L3 Lightcurves

Michael Nowak, MIT-CXC, October 5, 2007

Dither will introduce variability into lightcurves primarily in two manners. The first is via modulation of the fraction of the area that is on a chip as a function of time. This fraction vs. time is caculated by the `dither_region` tool. For a uniform response, the signal should be modulated in direct proportion to the modulation of the fractional area. The second is due to the extraction region moving among detector areas with substantially different responses. An obvious case is moving between a backside and frontside chip; however, moving across node boundaries might be sufficient to introduce a time-dependent signal. This variation is not accounted for in the `dither_region` tool (although the tool does divide the fractional area of the region into its component contribution from individual chips). Correcting for the latter effect would require, at a minimum, an average response for each energy band on each chip as a function of time. For the case of a near on-axis point source dithering between two chips, this perhaps could be simply two responses which are averaged in proportion to the fractional contribution to the extraction area from each chip (already provided by `dither_region`). For extended sources, and sources far off axis, it becomes a more difficult problem (potentially requiring extraction and time-dependent weighting of many repsonses). For this reason, I suggest that for the time being, we concentrate only on removing dither variability introduced by extraction area changes. This alone makes a substantial improvement to the calculated variability properties of sources (see my memo of Aug. 30, 2007).

The information provided by the `dither_region` tool is sufficient to provide the correction for the Kolmogorov-Smirnov/Kuiper tests as well as for the Gregory-Loredo lightcurve. The `dither_region` tool needs to be run once for each source, and once for the source's associated background region. These time vs. fractional area curves then need to be savved for use in these test. It would suffice to store these data in an extension added to the source and background lightcurve files. Along with the fractional area vs. time information, the files must also store the total area (in square pixels) for the source region and the background region at the value of fractional area = 1.

The procedure for correcting the variability tests is then as follows.

1. Run the `dither_region` tool on the source region, and then store the results.

   • It would suffice to store this information as an extension to the same file that stores the source lightcurve.

   • One could store the `areafraction` extension created by the `dither_region` tool. However, the only necessary components are the `TIME` and `FRACAREA` columns. (Storing the `AREA_CHIP_FRAC` column could be useful later on, should we wish to attempt to introduce response variations as well.)

2. Store the value of the area of the extraction region (in square pixels) for `FRACAREA` =1.

   • This value could be stored in the header of the extension that stores the `TIME` and `FRACAREA` columns.

3. Run the `dither_region` tool on the background region, and likewise store the `TIME`, `FRACREA`, and region size (in square pixels) results.

4. The background region information will not be directly used in the variability tests. It will be used in creating the Gregory-Loredo lightcurve for the background. The background region area will be employed by the user to scale the background lightcurve, should they wish to subtract it from the source lightcurve.

   • I suggest background subtraction of the lightcurve be a user choice, and not done automatically for the catalog.

5. For the Kolmogorov-Smirnov/Kuiper tests, source extraction region variability is accounted for in the initial set-up of the model (done via the `make_model` function in the `S-lang` code). Essentially, one is integrating the product of the good time intervals with the FRACAREA vs. TIME curve. The cumulative integral of this product is the cumulative distribution function against which we compare the data. The K-S/Kuiper test with these model changes is otherwise then run as normal.

   • I have attached `S-lang` code at the end of this document implementing this procudere, using interpolation and integration schemes from the `S-lang` GSL-module. Using the GSL-module is not a strict requirement. Any comparable interpolation/integration scheme would suffice.

   • The attached `S-lang` code also contains a rewrite of the "GTI correction" only part of the current `make_model` code. Mine is somewhat faster ($> 2\times$), and less prone to error.

6. As the Kolmolgorov-Smirnov/Kuiper tests are only used to assess variability, and not being used to create a lightcurve, there is no procedure here to be applied to the background lightcuve or regions.

7. As coded, the `glvary` tool already is applying GTI information, and is set-up to incorporate the FRACAREA information via the `eff` file input. This should suffice; however, one will need to use the times from the output lightcurve when generating the associated background lightcurve.

   • I believe the only major difference from my code and the `C`-code is in the definition of the lightcurve. I add a term for the no-subdivisions portion of the probability. This will only be relevant for low-probability values; however, as we are going down to 0.5 probabilities in the catalog, it's probably worth making this adjustment to `glvary`. That would be something for Arnold Rots to implement.

   • A stylistic difference is that for my final output lightcurve, I choose times at the beginnings of bin edges, whereas `glvary` seems to be choosing the middles of bins. As long as the calculations are done properly, this doesn't make any difference. However, this does make me realize that using the `glvary` output straight in a cross-correlation is a *bad* idea, since the time bins are not statistically independent. I'll discuss this more in a later memo on background flares.

   • The lightcurve output of the routine needs to be (and I believe is) counts/sec/fracarea, i.e., normalized to FRACAREA=1.

   • I have attached a revised version of my `S-lang` code for reference. It doesn't have the input/output features of `glvary`, nor all of the same toggles; however, it does give the proper output lightcurve (for a given set of times), and runs comparably fast as the `C`-code. It should be useful as a reference for any questions.

8. The `glvary` code should be applied to the background region, with its own TIME vs. FRACAREA information applied. However, in contrast to the source lightcurve, where the code determines the output times for the lightcurve, here one should force the background lightcurve to be evaluated on the source lightcurve times. This product is what should be written as the background lightcurve.

   • Note that only the times of the background lightcurve should be specified by the source lightcurve. `mmax`, the maximum number of partitions to use in creating the lightcurve, should not be (beyond the normal starting value). I.e., don't force the background lightcurve to be the same `mmax` as the source.

   • Again, this lightcurve is to be counts/sec/fracarea (i.e., normalized to FRACAREA = 1.

   • The background lightcurve is not to be subtracted from the source lightcurve. That is a job for the user. Specifically, the user should calculate it as: source lightcurve - (source region area/background region area)$\times$background lightcurve. This is the reason why we need to store the region areas.

• For sources and backgrounds that cover multiple chips, there may be some issues with consistency of the GTIs between the source and background. I would suggest that source and background share the same GTI, and that GTI be the intersection of the GTIs for each.

9. For the K-S/Kuiper tests and the Gregory-Loredo variability test, the TIME vs. FRACAREA curve only needs to be generated once for the source and once for the background. The same information is used for the four science bands and the integrated band. However, each science band and the integrated band will have the maximum partitions, mmax, and the times over which the lightcurve is evaluated, be individually determined for that band. The corresponding background lightcurve file then follows the specific results for the source lightcurve in that band.

    • Since all energy bands will share the same TIME vs. FRACAREA curve, and histogramming this is one of the more computationally expensive parts of the process, one can consider calculating it once, and then using it as a passed value for the subsequent runs. In my S-lang code, this is the gl_struct.aj piece. However, running that piece of code five times instead of once is probably not as much computation time as running dither_region once. Getting close, but probably still less. But if one were looking for an area to shave off more time, that would be it.

Here is my S-lang code for correcting make_model for the K-S/Kuiper test.

```
% To incorporate GTI and region area changes due to dither,
% all one really needs to do is add it to the definition of
% the model.  GTI are already incorporated into the model.
% Below we implement a slightly improved version of adding
% GTI only (exact same interface as the old make_model),
% and a newer version that also takes the output of
% the dither_region tool.

define make_model()
{

  variable t_event, time_lo, time_hi;

  (t_event,time_lo,time_hi) = ();

  variable mm    = length(t_event);
  variable modl = Double_Type[mm];

  variable ii,jj;
  variable nn;

  nn = length(time_lo);

  variable tot_gti = sum(time_hi - time_lo);
  variable frac = (time_hi - time_lo) / tot_gti;

% The following should replace the lines in the current code.
% It is more than 2X faster, and won't break if there happens
% to be an event time that falls between GTI boundaries
```

```
%%%

  variable cfrac = [0.,cumsum(frac)];

  _for jj (0,mm-1,1)
  {
    variable ifrac = max(where(time_lo <= t_event[jj]));
    if(time_hi[ifrac] < t_event[jj])
    {
       modl[jj]=cfrac[ifrac+1];
    }
    else
    {
       modl[jj]=cfrac[ifrac] +
                 (t_event[jj]-time_lo[ifrac])/tot_gti;
    }
  }

%%%

  return modl;
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                    %
% Now to incoporate dither_region results:           %
%                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% We're going to use the GSL interpolation routines,
% as for my take on the GLVary tool.  One could probably
% do this in a lot of ways - but the basic point is that
% the model is the integrated (region X gti) curve up to
% the event time, divided by the integral over the whole
% interval.  I.e., it goes from 0 to 1, and in exactly
% the same fashion as above if region = constant.

require("gsl");

define make_model_II()
{

  % ta,adt hold fractional area/deadtime corrections vs. time

  variable t_event, time_lo, time_hi, ta, adt;

  (t_event,time_lo,time_hi,ta, adt) = ();
```

```
variable lgti, icheck, ibad = Integer_Type[0];
lgti = length(time_lo);

% Merge fractional area vs. time with GTI information

icheck = where(ta < time_lo[0] or ta > time_hi[lgti-1]);
if(length(icheck)>0){ ibad = [ibad,icheck]; }

if(lgti>1)
{
   variable j;
   _for j (0,lgti-2,1)
   {
      icheck = where(ta>= time_hi[j] and ta<=time_lo[j+1]);
      if(length(icheck)>0){ ibad = [ibad,icheck]; }
lgti = length(time_lo);

% Merge fractional area vs. time with GTI information

icheck = where(ta < time_lo[0] or ta > time_hi[lgti-1]);
if(length(icheck)>0){ ibad = [ibad,icheck]; }

if(lgti>1)
{
   variable j;
   _for j (0,lgti-2,1)
   {
      icheck = where(ta>= time_hi[j] and ta<=time_lo[j+1]);
      if(length(icheck)>0){ ibad = [ibad,icheck]; }
   }
}

if( length(ibad) > 0 ) { adt[ibad]=0.; }

% Define the spline of the effective area curve, and get
% the integral over the whole GTI range.

variable spline_adt = interp_akima_init(ta,adt);
variable a_avg =
   interp_eval_integ(spline_adt,time_lo[0],time_hi[lgti-1]);

% Start setting up the model

variable mm = length(t_event);
variable modl = Double_Type[mm];

% Do integration of spline in neighboring points, then sum
```

```
  modl[0] = interp_eval_integ(spline_adt,time_lo[0],t_event[0]);

  _for j (1,mm-1,1)
  {
     modl[j] = interp_eval_integ(spline_adt,t_event[j-1],t_event[j]);
  }
  modl=cumsum(modl)/a_avg;

  return modl;
}
```

Attached below is my `S-lang` code for performing the Gregory-Loredo test and creating the associated lightcurve. It doesn't perform the $3-\sigma$ deviation test like `glvary` does, and it chooses a different binning for the output lightcurve, but otherwise it works more or less the same. This almost as fast as the `C`-code, and is probably about as fast as I can make it. It is designed to run in `ISIS`, and use the GSL-module, but there aren't that many external dependencies in it. It's a good reference to compare the `glvary` code against.

```
% GSL is for interpolating/integrating the effective area/deadtime

require("gsl");

% The GSL lngamma function is pretty fast, but tabulating lngamma
% is slightly faster, especially for multiple runs

#ifnexists fst_lngamma
   private variable log_sum = Double_Type[320001];
   log_sum[[0,1]] = [0.,0.];

   variable i;
   for(i=2; i<=320000; i++)
   {
      log_sum[i] = log_sum[i-1] + log(i);
   }

   public define fst_lngamma(i)
   {
      return log_sum[int(i-1)];
   }
#endif

% t = event times; tmin & tmax =  max & min time of lightcurve,
% mmax = maximum partition number in trial lightcurves
% ta, adt = effective area/deadtime times and values
% dodither = Toggle for applying dither correction to lightcurve
% thresh = helps determine truncation of partitioned lightcurve
```

```
define log_odds(t,tmin,tmax,mmax,thresh,dodither,ta,adt,nmult)
{
   if(dodither !=0)
   {
      variable inz = where(adt>0);
      variable adt_use = [inz[0]:inz[length(inz)-1]];

      ta = ta[adt_use];
      adt = adt[adt_use];

      variable na = length(adt);
      if(tmin < ta[0]){ tmin = ta[0]; }
      if(tmax > ta[na-1]){ tmax = ta[na-1]; }
   }

   % Only look at times within tmin & tmax

   t = t[where(t>=tmin and t<tmax)];

   variable j;
   variable n=length(t);
   variable dt_int = tmax - tmin;
   variable a_avg=1.;

   if(dodither !=0)
   {
      % Define the spline of the effective area curve

      variable spline_adt = interp_akima_init(ta,adt);
      a_avg = log(interp_eval_integ(spline_adt,tmin,tmax)/dt_int);

      % Do integration of spline in neighboring points, then sum

      variable iadt = Double_Type[na];
      _for j (1,na-1,1)
      {
         iadt[j] = interp_eval_integ(spline_adt,ta[j-1],ta[j]);
      }
      iadt=cumsum(iadt);

      % Define the spline of the integrated effective area curve

      variable spline_iadt = interp_akima_init(ta,iadt);
   }

   variable nj = Array_Type[int(mmax)-1];
   variable aj = @nj;
```

```
variable m=[2:int(mmax):1];
variable lods=Double_Type[int(mmax)-1];

variable lo,hi,im;

_for im (2,int(mmax),1)
{
   % Create grid for partitioning lightcurve into im bins

    lo = [0:im-1];
    hi = [1:im];
    lo = __tmp(lo)*(dt_int/im);
    hi = __tmp(hi)*(dt_int/im);

   if(dodither != 0 )
   {
      % For each partitioning, create average deadtime/effective
      % area per bin.  Dead bins are set equal to the average
      % effective area, so as not to contribute to the sum

      aj[im-2] = (interp_eval(spline_iadt,hi)
                  -interp_eval(spline_iadt,lo))*(im/dt_int);

      aj[im-2][where(aj[im-2]==0.)] = a_avg;
   }
   else
   {
      aj[im-2] = Double_Type[im]+1;
   }

   % For each lightcurve partition, the arrays of counts per bin

   nj[im-2] = histogram(t,lo,hi);

   % The odds ratio for each partitioning.  The nj[im-2]*()
   % term is removed if effective area variation is unimportant

   if(dodither != 0 )
   {
      lods[im-2] = sum( nj[im-2]*(a_avg-log(aj[im-2])) +
                        fst_lngamma(nj[im-2]+1) ) +
                   n*log(im) + fst_lngamma(im) - fst_lngamma(n+im);
   }
   else
   {
      lods[im-2] = sum( fst_lngamma(nj[im-2]+1) ) +
                   n*log(im) + fst_lngamma(im) - fst_lngamma(n+im);
   }
```

```
}

% This bit uses the return pieces from above to find what the
% maximum odds ratio is, and temporarily takes that out (lomax -
% a replacement for Arnold's bias parameter), and then like
% Arnold's code, truncates the number of lightcurve binnings kept

variable iw = where(lods==max(lods));
variable lomax = lods[min(iw)];

lods = exp(lods-lomax);
variable csum = cumsum(lods)/(m-1);

% Truncate the number of partitionings of the lightcurve

if(thresh < 0) thresh==0.;
variable imax = max(where(csum >= max(csum)/exp(thresh)));
iw = where(lods[[0:imax]]==max(lods[[0:imax]]));

% Return the probability (p), the odds ratio for each partioning
% (oratio), the log of the summed odds (lodds_sum), the # of
% partitions for the peak odds ratio (mpeak), the # of partitions
% for each (m), the histogrammed counts for each partitioning (nj),
% the integrated and averaged effective area for each partioning
%(aj, a_avg), and the used min/max times (tmin/tmax)

variable gl_struct=
  struct{p, oratio, lodds_sum, mpeak, mmax, m, nj,
         aj, a_avg, tmin, tmax, tlc, rate, erate};

% The # of partitions of the lightcurve with the highest odds ratio.

gl_struct.mpeak = min(iw)+2;

% Calculate the total probability of variability (p) and the odds
% ratio for each partitioning of the lightcurve (oratio).

variable msum = csum[imax];
variable lsum = log(msum) + lomax;
gl_struct.p = 1/(exp(-lsum)+1);
gl_struct.oratio = __tmp(lods)[[0:imax]]/
                         ((imax+1)*(msum+exp(-lomax)));

% The rest of the return values

gl_struct.lodds_sum = lsum;
gl_struct.mmax = imax+2;
gl_struct.m = __tmp(m);
```

```
gl_struct.nj = __tmp(nj);
gl_struct.aj = __tmp(aj);
gl_struct.a_avg = exp(a_avg);
gl_struct.tmin = tmin;
gl_struct.tmax = tmax;

% If nmult !=0, return a lightcurve with nmult*mmax bins

if(nmult !=0)
{
   variable tfrac = [0:nmult*(imax+2)-1]/(nmult*(imax+2)*1.);
   variable rate = Double_Type[nmult*(imax+2)];
   variable erate=@rate;

   % We might not have used all the data ...

   variable ntot=sum(gl_struct.nj[0]);

   variable nj_ii_k, oratio_ii, aj_ii_k, drate;

   % Best estimate of the lightcurve is calculated here.
   % Fractional area/deadtime correction is included.

   variable ii;
   _for ii (0,imax,1)
   {
      variable mloop=ii+2;
      variable k = int( tfrac*mloop );

      nj_ii_k=gl_struct.nj[ii][k];
      oratio_ii=gl_struct.oratio[ii];
      aj_ii_k = gl_struct.aj[ii][k];
      drate = (mloop/(ntot+mloop))*
              (__tmp(oratio_ii)*(nj_ii_k+1)/aj_ii_k);
      rate = __tmp(rate) + drate;
      erate = __tmp(erate) + (mloop/(ntot+mloop+1))*
              (__tmp(drate)*(__tmp(nj_ii_k)+2)/__tmp(aj_ii_k));
   }

   % Here I differ from Gregory&Loredo and Arnold. G-L only include
   % the variable part of the lightcurve (i.e., partitionings with
   % >=2 bins), reasonable for p~1. L3 goes down to p~0.6, therefore
   % the estimated constant lightcurve part should be included.

   rate = __tmp(rate) + (1.-gl_struct.p)/gl_struct.a_avg;
   erate = __tmp(erate) + (1.-gl_struct.p)/gl_struct.a_avg^2;
   erate = sqrt(__tmp(erate)-rate^2);
```

```
      gl_struct.tlc = __tmp(tfrac)*(tmax-tmin)+tmin;
      gl_struct.rate =
         __tmp(rate)*(ntot/(tmax-tmin));
      gl_struct.erate =
         __tmp(erate)*(ntot/(tmax-tmin));
   }

   return gl_struct;
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          %
%  Apply above to test file %
%                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

variable file,fp,t_event,mmax;
variable res;
variable tfrac,iw,k;

variable tmin = 7.2039524e7, tmax = 7.2141507e7;
mmax = int(min([(tmax-tmin)/50,3000]));

variable i=1;
file = sprintf("%04d",i);

% Read data file

t_event = fits_read_col("dither_sourceIII.fits","time");

% Read good time intervals

variable start_gti,stop_gti,lgti,icheck,ibad=Integer_Type[0];

(start_gti,stop_gti) =
   fits_read_col("dither_sourceIII.fits[GTI]","start","stop");
lgti = length(start_gti);

%  Read deadtime/fractional area vs. time

variable ta, adt;
(ta,adt) = fits_read_col("dither_sourceIII.frac","time","fracarea");

% Merge fractional area vs. time with GTI information

icheck = where(ta < start_gti[0] or ta > stop_gti[lgti-1]);
if(length(icheck)>0){ ibad = [ibad,icheck]; }
```

```
if(lgti>1)
{
   variable j;
   _for j (0,lgti-2,1)
   {
      icheck = where(ta>= stop_gti[j] and ta<=start_gti[j+1]);
      if(length(icheck)>0){ ibad = [ibad,icheck]; }
   }
}

if( length(ibad) > 0 ) { adt[ibad]=0.; }

tmin = min(ta);
tmax = max(ta)-tmin;
ta = ta-tmin;
t_event = t_event-tmin;

% Run the G-L test

variable res = log_odds(t_event,0,tmax,mmax,0.5,1,ta,adt,300);

% For p>0.9, plot and save the info.  (Note - I haven't added
% the additional logic by Arnold to calculate 3 sigma deviations
% of the lightcurve, as an additional variability check.)

%%%%%%%%%%%%%%%%%%%%%%%%%
%                       %
%  Print and Plot Stuff  %
%                       %
%%%%%%%%%%%%%%%%%%%%%%%%%

if(res.lodds_sum > log(9))
{
   variable ntot=sum(res.nj[0]);
   print("\nFile    Peak m  m_max  Total Counts  Log Odds");
   ()=printf("%4i    %4i    %4i    %7i    %9.2f\n\n",
             i,res.mpeak,res.mmax,int(ntot),res.lodds_sum);
   xrange;
   yrange(min([res.rate-2*res.erate,0.8*res.rate]),
          max([res.rate+2*res.erate,1.2*res.rate]));
   charsize(1.12);
   xlabel("\\frTime (sec)");
   ylabel("\\frCorrected Rate (cts/sec/area)");
   set_frame_line_width(3);
   set_line_width(2);
   linestyle(2);

   plot(res.tlc,res.rate+res.erate,2);
```

```
   oplot(res.tlc,res.rate-res.erate,2);

   linestyle(1);

   oplot(res.tlc,res.rate,1);
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          %
%  Compare to GLVARY tool  %
%                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#iffalse
   variable t_glv, r_glv, e_glv;

   (t_glv, r_glv, e_glv) = fits_read_col("gl_sourceIII_lc.fits",
                           "time","count_rate","count_rate_err");
   t_glv = t_glv - tmin;
   linestyle(2);
   oplot(t_glv, r_glv-e_glv, 5);
   oplot(t_glv, r_glv+e_glv, 5);
   linestyle(1);
   oplot(t_glv, r_glv, 4);
#endif

xrange(); yrange();
```