To: CSC Distribution

From: Frank Primini

Subject: Specifications for Combining Data from Multiple ObsIDs in Aperture Photometry

Date: September 13, 2019

---

1 Introduction

The purpose of this memo is to document the specifications for combining data from multiple observations in aperture photometry in *CSC 2.0*. These specifications were originally provided to Data Systems via e-mail and prototype code in June, 2015, but no formal specification was provided at the time. An up-to-date, but simplified version of the prototype code is available in https://github.com/fprimini/XAP/tree/python3/, and is provided here in Appendix A.

In *CSC 2.0*, aperture photometry is performed at the observation, stack, and master level. At the stack level, data from all valid observations comprising the stack are combined. At the master level, data from all unambiguous valid observations are combined into a master average. In addition, data from observations comprising each time-ordered or flux-ordered Bayesian Block are combined.

The original *CSC 2.0* aperture photometry specifications call for combining data from multiple observations through a process called 'chaining MPDFs', in which the marginalized posterior probability distribution (MPDF) for one observation was used as the prior for a subsequent one, with the observations ordered by increasing itensity (see http://cxc.cfa.harvard.edu/csc2/memos/files/Primini_aperture_photometry_specs.pdf). Although initial research on this technique was promising, it was soon realized that it greatly increased both the complexity and processing burden for photometry. First, it essentially doubled the number of photometry runs per source, since a combination of n observations would require a chain of $n-1$ additional runs. Moreover, if the source spectrum varied, the order of observations might be different in different bands. Finally, the order of observations for different sources in a multi-source bundle might be different, requiring the process to be repeated for each source in the bundle. For these reasons, it was decided to use a simpler approach, in which $Sherpa$ is used to fit a single intensity to data from multiple observations and $pyBLoCXS$ is used to estimate confidence intervals. In the remainder of this memo, I outline the steps required in this technique, compare its results to those of the current algorithm for the case of a single observation, and discuss some of its difficulties.

Symbol	Definition
x, y	Image Coordinates
X_i, Y_i	Source Position for source i
$psf(X_i,\, Y_i,\, x,\, y)dxdy$	Telescope Point Spread Function at location x, y for a source at $X_i,\, Y_i$.
R_i	Source Aperture for source i
R_b	Background Aperture
Ω_i	Area of Source Aperture for source i (e.g. $pixel^{\,2}$)
Ω_b	Area of Background Aperture
E_i	Average Exposure Map Value in Source Aperture i
E_b	Average Exposure Map Value in Background Aperture
r_i	Ratio of Background to Source Aperture, $r_i = \Omega_b/\Omega_i$
C_i	Total Counts in Source Aperture i
B	Total Counts in Background Aperture
s_i	Net Source Counts for source i
b	Background Density (e.g. $counts - pixel^{-2}$)
f_{ij}	Fraction of PSF for source j enclosed in R_i, e.g., $\int_{R_i} psf(X_j,\, Y_j,\, x,\, y)dxdy$
g_i	Fraction of PSF for source i enclosed in R_b, e.g., $\int_{R_b} psf(X_i,\, Y_i,\, x,\, y)dxdy$
θ_i	Expected total counts in Source Aperture i: $\theta_i \;=\; \sum_{j=1}^{n}\; f_{ij}s_j \;+\; \Omega_i b$
ϕ	Expected total counts in Background Aperture: $\phi \;=\; \sum_{i=1}^{n}\; g_i s_i \;+\; \Omega_b b$

Table 1: Symbols and Definitions

2 Aperture Photometry for a Single Observation

2.1 Point Estimate for Source Intensities

I adopt the symbols and definitions from Table 1 in Primini_aperture_photometry_specs.pdf, repeated here in Table 1. In the context of fitting in *Sherpa*, the raw aperture counts may be considered a 1-D dataset with $(n + 1)$ elements, where n is the number of sources in the bundle, namely,

$$\mathbf{D} = \{C_i, B\} \tag{1}$$

Similarly, the model may be defined as a vector of the same dimensionality, where

$$\mathbf{M} = \{\theta_i, \phi\} \tag{2}$$

with model parameters

$$\mathbf{P} = \{s_i, b\}. \tag{3}$$

For a single observation, the number of parameters is the same as the number of elements in **D**, so there should be (at most) a single solution to the fit. If one of the *Sherpa* Poisson log-likelihood functions is used as the fit statistic, the result is essentially the Maximum-Likelihood solution for the source and background intensities[1]. These steps are carried out in lines $300 - 345$ in the function **run_mcmc** in Section A.1.

[1] In practice, this may not be true for some pathological cases in which Maximum-Likelihood values for one or more source intensity are negative. Valid source intensity parameter ranges in the *Sherpa* fit are forced to be positive.

In fact, it isn't necessary to use $Sherpa$ to determine the Maximum-Likelihood solution, since that may be calculated directly from the aperture counts, plus auxiliary exposure map, psf fraction, and aperture area data (see e.g. Equations 8 & 9 in Primini & Kashyap 2014, and lines $540 - 562$ in the prototype code in Section A.1). Indeed, the solution so obtained is used to define parameter guesses and ranges prior to the fit (see line 323 in Section A.1). However, recasting the problem as a $Sherpa$ fit does provide a convenient starting point for using the MCMC techniques in $pyBLoCXS$ to sample the joint posterior probability distribution (JPDF) for source intensities. It is expected that this will be more robust and less memory-intensive than mapping the JPDF on an $(n + 1)$-dimensional hyper-cube, as described in Section 2.1 of Primini_aperture_photometry_specs.pdf.

2.2 Confidence Intervals

As before, confidence intervals in intensity for each source in the bundle are derived from marginalized posterior probability distributions (MPDFs) for each source, by integrating symmetrically from the mode of the distribution until the desired confidence level is reached. However, the MPDFs themselves are generated from MCMC samples (draws) of the joint posterior probability distribution, rather than from numerical integration over all but one dimension of an $(n + 1)$-dimensional JPDF hyper-cube, as described in Primini_aperture_photometry_specs.pdf. This is illustrated in lines $540 - 562$ of the prototype code in Section A.1, where I have assumed default non-informative priors for each parameter (subject only to non-negativity). Each draw consists of a set of (s_i, b) parameter values determined by the MCMC algorithm to accurately sample the JPDF. Five thousand draws are generated, of which the first 100 are discarded.

Modes and confidence bounds for each parameter are estimated from uniformly-sampled MPDF arrays derived from the distribution of draws for that parameter. The range of samples is estimated from the Maximum-Likelihood solution to be $s_{ML} \pm 5\sigma_{ML}$. In the prototype code in Section A.1, I assume that the functional form of the MPDFs is a γ distribution, namely,

$$MPDF(s) = \frac{\beta^{\alpha} s^{\alpha-1} e^{-\beta s}}{\Gamma(\alpha)}, \tag{4}$$

with parameters α and β for the posterior γ distributions estimated from the draws $\{s_i\}$ using the relations

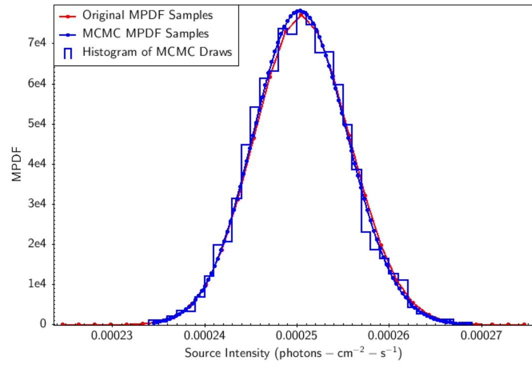
$$\alpha = mean(\{s_i\})^2/var(\{s_i\}) \tag{5}$$

$$\beta = mean(\{s_i\})/var(\{s_i\}). \tag{6}$$

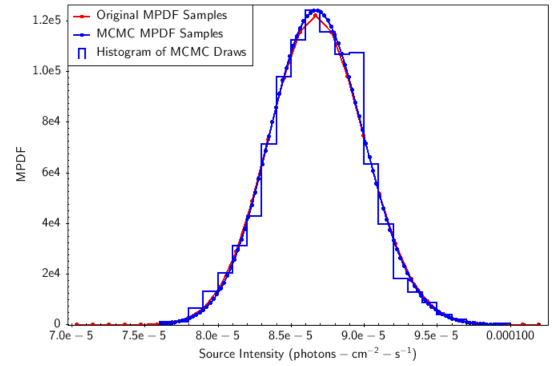
My reasoning is that the γ distribution is the conjugate prior for for Poisson likelihoods and its use leads to posterior distributions of the same functional form (Raiffa & Schlaifer, 1961). Non-informative priors may be considered a special case of a γ distribution for $\alpha \to 1$ and $\beta \to 0$.

Alternatively, a non-parametric smoothing function, such as a kernel density estimator, may be used to derive MPDFs from the distribution of draws.

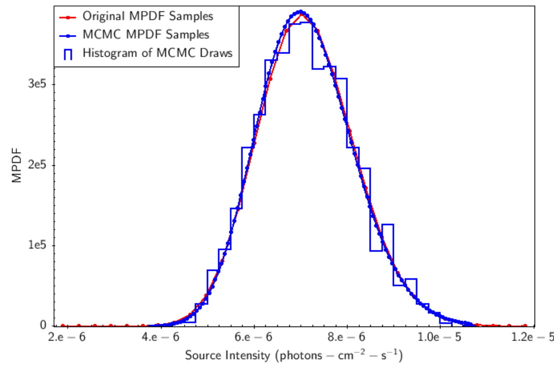
In Figure 1, I compare the MPDFs derived from numerical integration of the JPDF hyper-cube of the original specifications with the distribution of draws and MPDFs from the MCMC implementation, for sources in the four-source bundle discussed in Figure 2 of Primini & Kashyap (2014). For all cases, the MCMC MPDFs agree well with the distributions of draws, and the hyper-cube and MCMC implementations are in good agreement.



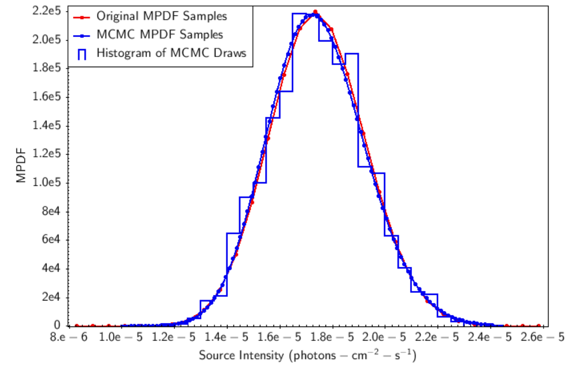
(a)



(b)



(c)



(d)

Figure 1: MPDFs and distributions of draws for sources r0115 (a), r0116 (b), r0123 (c) and r0150 (d) from Figure 2 in Primini & Kashyap (2014). Data, MPDFs and draws may be found in the **Example** and **MCMC_Example** directories in https://github.com/fprimini/XAP/tree/python3/.

3 Combining Data from Multiple Observations

This is accomplished by treating aperture data from each observation as a separate 1-D dataset, given by Equation 1, with models defined by Equation 2. A single $Sherpa$ fit is then performed simultaneously on all datasets, linking all source intensity parameters, but allowing separate background parameters for each observation. Parameter ranges for source intensities should encompass all of the parameter ranges in the individual observations. Initial guesses for source intensities should be taken from the longest exposure observation (TBR). The results of the fit are then used as a starting point in the $pyBLoCXS$ MCMC process to generate MPDFs and confidence intervals as in Section 2.2.

In Figure 2, I show an example of four simulated datasets of a single 10-count $ACIS-I$ source. Each dataset was analysed separately according to the procedures in Section 2 and the combined datasets were then analysed with the procedures in this section. The results are shown in Figure 3.

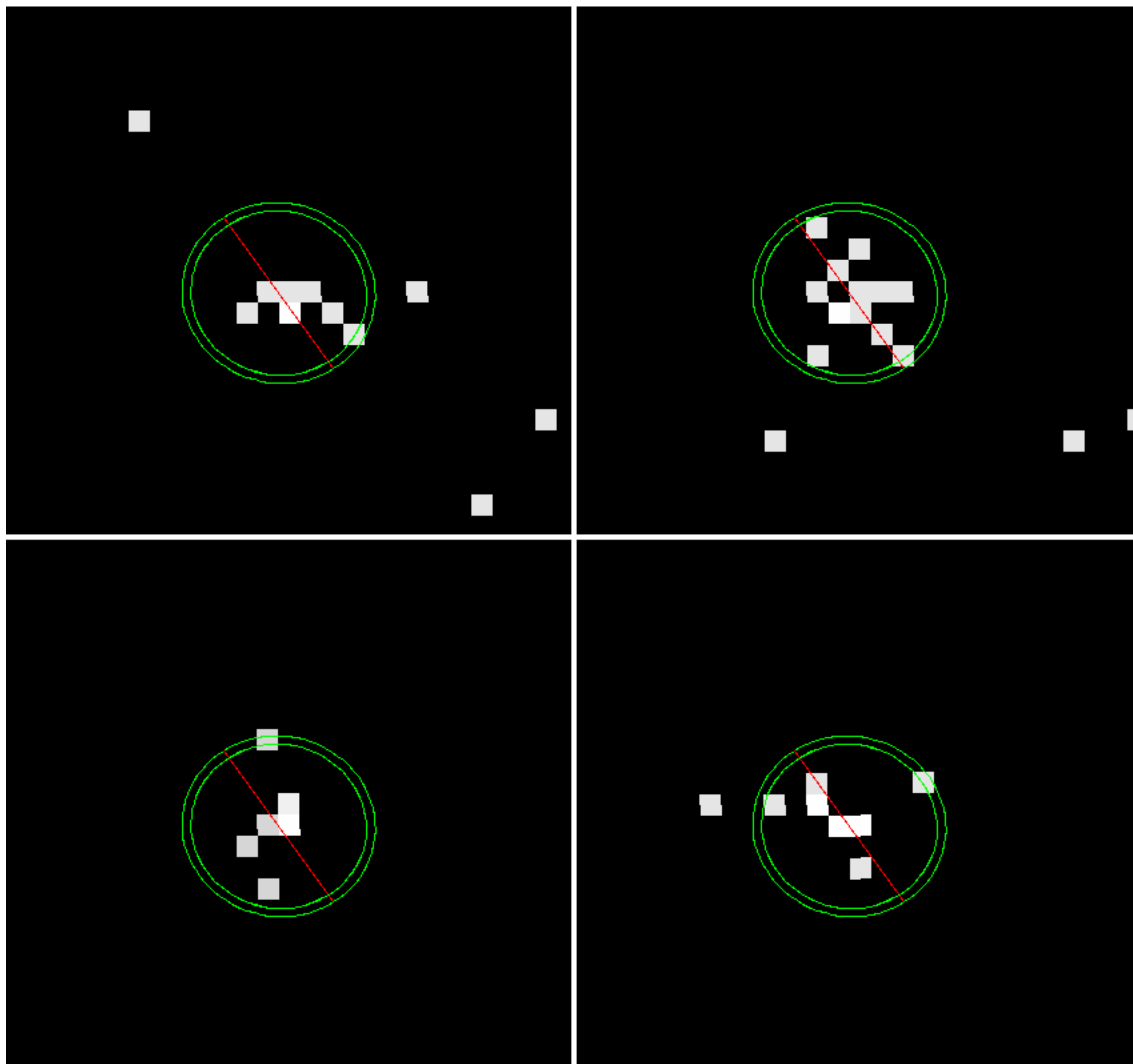


Figure 2: Four simulated $ACIS - I$ images with a 10 count source.

Results for the combined datasets include a mode near the average of the modes for the individual MPDFs, and a distribution width narrower than those for three of the individual distributions. A log of the analysis steps is given in Appendix refapp:sherpalog.

4 Potential Problems

Simultaneous fitting of single intensities to data from multiple multiple observations works best when, as in Section 3, the intensities in the contributing observations share the same parent distribution. This will perforce be true when combining data in Bayesian Blocks, since the Bayesian Blocks algorithm essentially forces that condition. However, in stack or master averages, if the source is sufficiently variable, it may be difficult to fit a single intensity to all datasets or to adequately sample the JPDF.

In such cases, experience shows that certain steps may help to mitigate the effects. First, if the covariance matrix (used to set the scale for the MCMC sampling) returned from the *Sherpa*

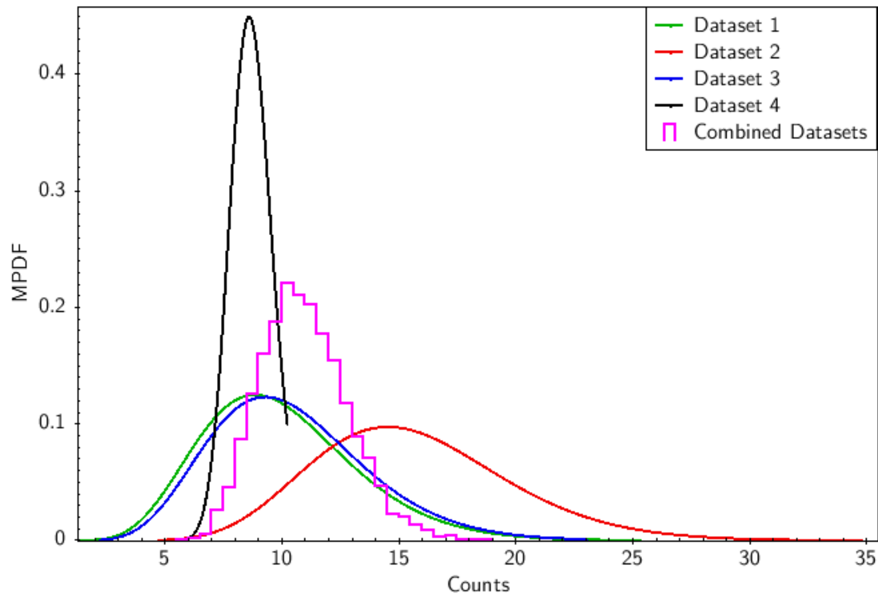


Figure 3: MPDFs for individual datasets and distribution of draws for the combined set, for the datasets shown in Figure 2.

fit is poorly determined, e.g., has negative diagonal elements, it may be replaced with values determined from the more accurate (but computationally more expensive) *Sherpa* **conf** function. If the MCMC chain fails to converge, it may be possible to force convergence by restricting the parameter space of the model by freezing all background parameters at their fitted values from the individual observations. At the master average level, all parameters except the intensity of the master source may be frozen at their individual observation values.

References

Primini, F. A., & Kashyap, V. L. 2014, ApJ, 796, 24

Raiffa, H. & Schlaifer, R. 1961, Applied Statistical Decision Theory 1st Edition (Cambridge, MA, MIT Press)

A Prototype Code

A.1 xap_mcmc.py

```python
#! /usr/bin/env python

# 06/14/2012  Updated to run under CIAO 4.4. Scipy and Pyfits dependencies removed.
# 06/19/2012  Updated to use akima interpolation provided by Christoph Gohlke <http://www.lfd.uci.edu/~gohlke/>
# 09/24/2012  Updated to remove _akima c library in akima interpolation routine.
# 06/13/2014  Updated to allow separate Gamma Distribution alpha/beta parameters via stacks for each source
#             and background
# 06/13/2014  Updated to allow command line parameter input
# 08/23/2019  Updated to python 3
# 08/26/2019  Clean up and rename to xap_mcmc.py. Incorporate relevant functions from xap_funs.py and xap_mcmc_funs.py and
#             deprecate those files (at least for this code).

import sys

def info(type, value, tb):
    if hasattr(sys, 'ps1') or not sys.stderr.isatty():
        # we are in interactive mode or we don't have a tty-like
        # device, so we call the default hook
        sys.__excepthook__(type, value, tb)
    else:
        import traceback, pdb
        # we are NOT in interactive mode, print the exception...
        traceback.print_exception(type, value, tb)
        print
        # ...then start the debugger in post-mortem mode.
        pdb.pm()

sys.excepthook = info

from pycrates import *
from numpy import *
from paramio import *
from region import *
from crates_contrib.utils import *
import math as m

###############################################################################

def get_F_C_exp(evtfile,sregs,breg,psfs,exps,exposure):
    """
    Inputs:
    evtfile    Event list file name (used to determine counts in regions)
    sregs      List of source region fits file names
    breg       Background region fits file name
    psfs       List of source psf fits image file names

    Compute array of psf fractions F and vector of total counts C, such that
    F[i,j] is PSF fraction of source j in source region i
    C[i] is total counts in source region i
    Here, the last source region is actually the background region, so
    F[n,j] is the PSF fraction of source j in the background region and
    C[n] is the total counts in the background region.
    Array F and vector C are returned.

    In this version, observation exposure is accounted for, either via exposure maps
    stack exps (one per source/background region) or header keyword.

    """

    # First determine sizes of F and C:

    ldim=len(sregs)
    ndim=ldim+1

    C=zeros(ndim)

    # If no psfs are provided, assume source ecf=1

    F=identity(ndim)

    # Now build C. First the source regions:

    for i in arange(0,ldim):
        evtfilter='{}[sky=region({})]'.format(evtfile,sregs[i])
        evts=read_file(evtfilter)
        crtype=get_crate_type(evts)
        if crtype == 'Table':
            C[i]=len(copy_colvals(evts,0)) # assuming event list has at least 1 column
        if crtype == 'Image':
            C[i]=sum(copy_piximgvals(evts))
        evts = None

    # and now the background region:

    evtfilter='{}[sky=region({})]'.format(evtfile,breg)
```

```
86      evts=read_file(evtfilter)
87      crtype=get_crate_type(evts)
88      if crtype == 'Table':
89          C[ldim]=len(copy_colvals(evts,0)) # assuming event list has at least 1 column
90      if crtype == 'Image':
91          C[ldim]=sum(copy_piximgvals(evts))
92      evts = None
93
94      # Next, build F. If psfs are specified, use them to generate the ecf's
95
96      if len(psfs)>0 :
97
98          # All but the last row and all but the last column of F contain
99          # the ecf's of source j in region i:
100
101         for i in arange(0,ldim):                                    # row loop
102             for j in arange(0,ldim):                                # column loop
103                 imgfilter='{}[sky=region({})]'.format(psfs[j],sregs[i])
104                 imgcr=read_file(imgfilter)
105                 F[i,j]=sum(copy_piximgvals(imgcr))
106                 imgcr = None
107
108         # All but the last column of the last row of F contain the ecf's of
109         # source j in the background region:
110
111         for j in arange(0,ldim):
112             imgfilter='{}[sky=region({})]'.format(psfs[j],breg)
113             imgcr=read_file(imgfilter)
114             F[ldim,j]=sum(copy_piximgvals(imgcr))
115             imgcr = None
116
117     # The last column in F contains region areas. All but the last are source regions:
118
119     for i in arange(0,ldim):
120         F[i,ldim]=regArea(regParse('region({})'.format(sregs[i])))
121
122     # And the last row, last column entry is the background region area.
123
124     F[ldim,ldim]=regArea(regParse('region({})'.format(breg)))
125
126     # Finally, modify by exposure. If exps are specified, compute average map value in
127     # each region:
128
129     ereg = ones(ndim)
130     if len(exps) > 0 :
131
132         # average expmap in each source region
133
134         for i in arange(0,ldim):
135             imgfilter = '{}[sky=region({})]'.format(exps[i],sregs[i])
136             imgcr     = read_file(imgfilter)
137             evals     = copy_piximgvals(imgcr)
138             enums     = evals.copy()
139             enums[enums>0.0]=1.0
140             ereg[i]   = sum(evals)/sum(enums)
141             imgcr = None
142
143         # Average expmap in background region
144
145         imgfilter = '{}[sky=region({})]'.format(exps[ldim],breg)
146         imgcr     = read_file(imgfilter)
147         evals     = copy_piximgvals(imgcr)
148         enums     = evals.copy()
149         enums[enums>0.0]=1.0
150         ereg[ldim]   = sum(evals)/sum(enums)
151         imgcr = None
152
153     # otherwise, use exposure from header for all regions
154
155     else:
156         ereg = ereg*exposure
157
158     F = F*ereg.reshape(ndim,1)
159
160     return F,C
161
162 ##############################################################################
163
164 def get_s_sigma(F,C):
165     """
166     Solve matrix equation C = F dot s for source intensity vector s.
167
168     Inputs:
169     F[i,j]      Array of encircled energy fractions and region areas
170     C[i]        Vector of region counts
171
172     Output:
173     s[i]        Vector of MLE estimates of source and background intensities
174     sigma_s[i] Vector of errors on MLE estimates, assuming Gaussian statistics
175
176     1/30/2014
177     Change sigma calculation to use covariance matrix method
```

```
178
179     5/5/2014
180     Go back to old propagation of errors technique to calculate sigmas
181     """
182
183     import numpy.linalg as la
184
185     # Solve equation by inverting matrix F:
186
187     Finv = la.inv(F)
188
189     s = dot(Finv,C)   # dot is matrix multiplation
190
191     # To get errors, need to square Finv:
192
193     Finv2 = Finv*Finv
194
195     sigma_s = sqrt(dot(Finv2,C))
196
197     return s,sigma_s
198
199
200 ###########################################################################################
201
202 class Apertures:
203     """
204     Class of user-defined models to compute model counts in source or
205     background apertures. Model parameters are source and background
206     intensities. Data are raw counts in apertures. The vector of model
207     counts is computed from the vector of intensities by application of
208     the ECF/Exposure matrix F (see eq. 7 and Table 1 of Primini & Kashyap,
209     2014, ApJ, 796, 24.)
210
211     Methods:
212     __init__(self,F)
213       define F matrix
214     __call__(self, params,iap)
215       compute model counts for vector aperture for F and model intensities given
216       in params array.
217     Attributes:
218       Print F.
219     """
220
221     def __init__(self,F):
222         self.F=F
223
224     def __call__(self,params,iap):
225         F=self.F
226         nelem=len(params)
227         mvec=zeros(nelem)
228         for i in range(nelem):
229             for j in range(nelem):
230                 mvec[i]=mvec[i]+F[i][j]*params[j]
231         return mvec
232
233     def attributes(self):
234         print ('ECF/Exposure Matrix:')
235         print (self.F)
236         print ('Length of F:')
237         print (len(self.F))
238
239 ###############################################################################################
240
241 class Gamma_Prior:
242     """
243     Compute Gamma Prior Distribution intensity vector, using alpha, beta specified in attributes
244     The prior distribution unnormalized and is defined as gp(s) = s**(alpha-1)*exp(-beta*s)
245
246     Methods:
247
248     __init__(self,alpha,beta)
249         initialize alpha, beta
250
251     __call__(self,s)
252         returns array of gp for input array of s values
253
254     __attributes(self)
255         print alpha and beta
256     """
257
258     def __init__(self,alpha,beta):
259         self.alpha = float(alpha)
260         self.beta  = float(beta)
261
262     def __call__(self,s):
263
264         # return a flat prior for alpha=1, beta=0
265
266         if(self.alpha==1.0 and self.beta==0.0):
267             return ones(len(s))
268
269         # Otherwise, evaluate full Gamma Distribution to avoid overflows
```

```
270
271             return exp(self.alpha*log(self.beta)+(self.alpha-1.0)*log(s)-self.beta*s-m.lgamma(self.alpha))
272
273         def attributes(self):
274             print ('Gamma Prior Attribute alpha:\t{:f}'.format(self.alpha))
275             print ('Gamma Prior Attribute beta:\t{:f}'.format(self.beta))
276
277 ########################################################################################
278
279
280 def run_mcmc(F,C,s,sigma_s,ndraws,nburn,scale):
281
282     # Wrapper for Sherpa commands to get draws
283     #
284     # Inputs:
285     #   F       - ECF/Exposure Matrix
286     #   C       - Raw aperture counts vector
287     #             Related to vector of model intensities for source
288     #             and background M by matrix equation
289     #             C = F x M
290     #   s       - MLE intensities vector
291     #   sigma_s  Errors on s
292     #   ndraws - Number of draws
293     #   nburn  - Number of draws to skip at start of sampling
294     #   scale  - Initial scale for covariance matrix in get_draws
295     #
296     # Outputs:
297     #   parnames,stats,accept,params - parameter names and vectors from
298     #   get_draws with first nburn samples eliminated
299
300     import sherpa.astro.ui as shp
301
302     # Create an instance of the Apertures class and load it as the user model
303
304     apertures = Apertures(F)
305     shp.load_user_model(apertures,"xap")
306
307     # Make a dummy independent variable array to go with C, and load it and C
308     # as the dataset
309
310     ix=arange(len(C))
311     shp.load_arrays(1,ix,C,shp.Data1D)
312
313     # Define parameter names. The last element in C is always the background.
314
315     parnames=[]
316     for i in range(len(C)-1):
317         parnames.append('Source_{}'.format(i))
318     parnames.append('Background')
319
320     # and add user pars, using s, sigma_s to establish guesses and bounds for parameters.
321     # Offset values of s by 5% to force re-fit for maximum likelihood.
322
323     shp.add_user_pars("xap",parnames,1.05*s,parmins=1.05*s-5.0*sigma_s,parmaxs=1.05*s+5.0*sigma_s)
324
325     # Set model, statistic, and minimization method
326
327     shp.set_model(xap)
328     shp.set_stat("cash")
329     shp.set_method("moncar")
330
331     # Finally, run fit and covar to set bounds for get_draws
332
333     # First, set hard lower limit for source intensity to avoid negative values
334
335
336     for i in range(len(xap.pars)):
337         xap.pars[i]._hard_min=0.0
338
339     import logging
340     logger = logging.getLogger("sherpa")
341
342 #     logger.setLevel(logging.ERROR)
343
344     shp.fit()
345     shp.covar()
346
347
348     # Check that covar returns valid matrix:
349
350     cvmat = shp.get_covar_results().extra_output
351     if any(isnan(cvmat.reshape(-1))) or any(diag(cvmat)<0.0):
352         print ("WARNING: covar() returned an invalid covariance matrix. Attempting to use conf().")
353         # Try to use conf() to set diagonal elements of covar matrix. If that doesn't work, use sigma_s
354         shp.conf()
355         conf_err=array(shp.get_conf_results().parmaxes)
356         if not all(conf_err>0.0):
357             print ("WARNING: conf() returned invalid errors. Using MLE errors.")
358             conf_err=sigma_s
359         cvmat = diag(conf_err*conf_err)
360
361     shp.get_covar_results().extra_output=cvmat
```

```
362        shp.set_sampler_opt('scale',scale)
363        stats,accept,intensities=shp.get_draws(niter=ndraws)
364
365        # and return all but the first nburn values. If nburn>=ndraws,
366        # return a single draw.
367
368        nburn1=min(nburn,ndraws-1)
369        return parnames,stats[nburn1:],accept[nburn1:],intensities[:,nburn1:]
370
371    ##################################################################################################
372
373    def write_draws(filename,parnames,stats,accept,intensities):
374
375        # Use crates to save draws in a fits file
376
377        # Inputs:
378        # filename - output draws file name
379        # parnames - names of source/backgrounds
380        # stats    - vector of stats from get_draws
381        # accept   - vector of draws acceptances
382        # intensities - 2-d table of draws for each aperture
383
384        tab=TABLECrate()
385        tab.name="Draws"
386        add_colvals(tab,'Stat',stats)
387        add_colvals(tab,'Accept',accept)
388
389        for i in range(len(parnames)):
390            add_colvals(tab,parnames[i],intensities[i])
391
392        ds=CrateDataset()
393        ds.add_crate(tab)
394        ds.write(filename,clobber=True)
395
396        return
397
398    ##################################################################################################
399
400    def write_draws_mpdfs(filename,parnames,intensities,ndrmesh):
401
402        # Estimate mpdfs from draws, and use crates to save mpdfs in a fits file
403
404        # Inputs:
405        # filename - output draws mpdfs file name
406        # parnames - names of source/backgrounds
407        # intensities - 2-d table of draws for each aperture
408        # ndrmesh  - number of grid points in mpdfs
409
410        cds=CrateDataset()
411        for i in range(len(parnames)):
412            sdraws=intensities[i]
413
414            # Compute Gamma Dist alpha, beta from mean and variance
415            # of intensity values in draws
416
417            smean = sum(sdraws)/len(sdraws)
418            svar  = sum(sdraws*sdraws)/len(sdraws) - smean*smean
419            alpha = smean*smean/svar
420            beta  = smean/svar
421
422            smin  = max(min(sdraws),1.0e-10)        # should never happen, but just in case, make sure sdraws>0
423            smax  = max(sdraws)
424            ds    = (smax-smin)/ndrmesh
425            sgrid = arange(smin,smax,ds)
426
427            # First Gamma Dist
428
429            gps    = Gamma_Prior(alpha,beta)
430            draws_mpdf = gps(sgrid)
431
432            tab=TABLECrate()
433            tab.name=parnames[i]+" "+"Gamma_Dist"
434            add_colvals(tab,'Inten',sgrid)
435            add_colvals(tab,'MargPDF',draws_mpdf)
436            set_key(tab,'alpha',alpha)
437            set_key(tab,'beta',beta)
438            set_key(tab,'smean',smean)
439            set_key(tab,'svar',svar)
440            set_key(tab,'Stepsize',ds)
441            cds.add_crate(tab)
442
443        cds.write(filename,clobber=True)
444
445        return
446
447    ##########################################################################################
448
449    def usage():
450        print ("Usage: xap_mcmc.py @@<CIAO-style parameter file>")
451        print ("Default parameter file is xap_mcmc.par")
452        return
453
```

```
454 #############################################################################################
455
456 def main(argv):
457
458     # Get Inputs from parameter file:
459
460     try:
461         fp=paramopen("xap_mcmc.par","wL",argv)
462     except:
463         usage()
464         sys.exit(2)
465
466     evtfile=pget(fp,"infile")
467     drawsfile=pget(fp,"drawsfile")
468     drawsmpdffile=pget(fp,"drawsmpdffile")
469     breg=pget(fp,"breg")
470     srcstack=pget(fp,"srcstack")
471     scr=read_file(srcstack)
472     sregs=get_colvals(scr,0)
473
474     # Read psfstack and test for null values
475
476     psfs = array([])
477     psfstack=pget(fp,"psfstack")
478     try:
479         pcr=read_file(psfstack)
480         psfs=get_colvals(pcr,0)
481     except:
482         print( "Unable to find PSF files.\nSetting source region ecfs to 1.0\n and background ecf to 0.0\n")
483
484     # Read expstack and test for null values
485
486     exps = array([])
487     expstack=pget(fp,"expstack")
488     try:
489         ecr=read_file(expstack)
490         exps=get_colvals(ecr,0)
491     except:
492         print( "Unable to find exposure maps.\nOutput units will be \"counts/sec\".\n")
493
494     #if len(sregs) != len(psfs):
495     #    print( "Error: Source and PSF Stacks Inconsistent")
496
497     CL_desired    = pgetd(fp,"CL")
498     intenstack = pget(fp,"intenstack")
499     nmesh         = pgeti(fp,"nmesh")
500     ndraws        = pgeti(fp,"ndraws")
501     nburn         = pgeti(fp,"nburn")
502     scale         = pgetd(fp,"scale")
503     ndrmesh       = pgeti(fp,"ndrmesh")
504     clob          = pgetb(fp,"clobber")
505     verb          = pgeti(fp,"verbose")
506
507     paramclose(fp)
508
509     # Get EXPOSURE from event list header
510
511     exposure = 1.0
512     try:
513         exposure= get_keyval(read_file(evtfile),'exposure')
514     except:
515         print( "Unable to find EXPOSURE keyword in header to {}\nOutput units will be \"counts\".".format(evtfile))
516
517     # Get MJD_OBS from event list header
518
519     mjdobs = -1.0
520     try:
521         mjdobs= get_keyval(read_file(evtfile),'MJD_OBS')
522     except:
523         print( "Unable to find MJD_OBS keyword in header to {}\nObservation Time will be set to -1.0".format(evtfile))
524
525     if verb>0:
526         print( "Getting Events from {}\n".format(evtfile))
527         print( "Exposure:\t{}".format(exposure))
528         print( "Source Regions:")
529         for i in arange(0,len(sregs)):
530             print( sregs[i])
531
532         print( "\nBackground Regions:")
533         print( breg)
534         print( "\nPSF Images:")
535
536         for i in arange(0,len(psfs)):
537             print( psfs[i])
538         print( "\n")
539
540     # Now set up F and C arrays:
541
542     F,C = get_F_C_exp(evtfile,sregs,breg,psfs,exps,exposure)
543
544     number_of_params = len(C)          # This is source plus background
545     number_of_sources = number_of_params -1        # Number of sources only
```

```
546
547         if verb>0:
548             for i in arange(0,number_of_sources):
549                 print( "Counts in Region {}:\t\t{}".format(i,C[i]))
550
551             print( "\nCounts in Background Region:\t{}".format(C[number_of_sources]))
552
553             print ("\nPSF Fractions Matrix F:")
554             for i in arange(0,number_of_params):
555                 for j in arange(0,number_of_sources):
556                     print("{:.4e}".format(F[i][j]),end='  ')
557                 print("{:e}".format(F[i][number_of_sources]))
558             print ("\n")
559
560         # Solve for MLE intensities and errors:
561
562         s,sigma_s = get_s_sigma(F,C)
563
564         # Finally, run mcmc:
565
566         print( "Running MCMC...")
567
568         parnames,stats,accept,intensities=run_mcmc(F,C,s,sigma_s,ndraws,nburn,scale)
569
570         write_draws(drawsfile,parnames,stats,accept,intensities)
571
572         # Write equivalent of mpdfs files, but using draws to estimate mpdfs
573
574         write_draws_mpdfs(drawsmpdffile,parnames,intensities,ndrmesh)
575
576     if __name__ == "__main__":
577         main(sys.argv)
```

A.2 xap_mcmc.par

```
infile,f,a,"acisf01575_001N001_evt3.fits.gz[energy=500:7000]",,,"Input event list or image"
drawsfile,f,a,"mcmc_r0115_r0116_r0123_r0150_draws.fits",,,"Output binary fits table of MCMC samples for all intensities"
drawsmpdffile,f,a,"mcmc_r0115_r0116_r0123_r0150_mpdfs.fits",,,"Output binary fits table of pdfs derived from MCMC samples"
srcstack,f,a,"r0115_r0116_r0123_r0150_srcs.lis",,,"Source Regions"
breg,f,a,"new_acisf01575_001N001_r0116_bkgreg3.fits",,,"Background Regions"
psfstack,f,a,"r0115_r0116_r0123_r0150_psfs.lis",,,"PSF Files"
expstack,f,a,"r0115_r0116_r0123_r0150_regexps.lis",,,"EXP Files"
CL,r,h,0.68269,,,"Desired Confidence Level"
intenstack,f,a,"none",,,"Source Intensities and Variances for Gamma-Priors"
nmesh,i,h,100,,,"Number of mesh points per source"
nburn,i,h,100,,,"Number of initial draws to skip"
scale,r,h,1,,,"Initial scaling factor for covar matrix"
ndraws,i,h,5000,,,"Number of draws"
ndrmesh,i,h,100,,,"Number of mesh points per source for draws mpdfs"
verbose,i,h,1,0,5,"Debug Level(0-5)"
clobber,b,h,yes,,,"Overwrite existing outputs?"
mode,s,h,"l",,,
```

B Sherpa Log

(Individual routines in the module **xap_mcmc_funs** may be found in Section A)

```
calf -13: sherpa
-----------------------------------------------------
Welcome to Sherpa: CXC's Modeling and Fitting Package
-----------------------------------------------------
CIAO 4.7 Sherpa version 1 Thursday, December 4, 2014

sherpa-1> import sys
sherpa-2> sys.path.append('/Users/fap/bin')
sherpa-3> import xap_mcmc_funs as xap

sherpa-5> from numpy import *

sherpa-7> F1=array([[8.83838290e-01,   1.24255379e+01],[7.11544836e-02,   3.00000000e+02]])
sherpa-8> C1=array([8.,4.])
sherpa-9> ix1=array([0,1])
sherpa-10> s1=array([9.9756e+00,1.4316e-02])
sherpa-11> sigma_s1=sqrt(array([1.1567e+01,5.6354e-05]))

sherpa-13> F2=array([[8.83838290e-01,   1.24255379e+01],[7.11544836e-02,   3.00000000e+02]])
sherpa-14> C2=array([13.,3.])
sherpa-15> ix2=array([0,1])
sherpa-16> s2=array([1.5599e+01,9.9108e-03])
sherpa-17> sigma_s2=sqrt(array([1.7561e+01,4.3555e-05]))

sherpa-18> F3=array([[8.83838290e-01,   1.24255379e+01],[7.11544836e-02,   3.00000000e+02]])
sherpa-19> C3=array([8.,2.])
sherpa-20> ix3=array([0,1])
sherpa-21> s3=array([9.9643e+00,7.9220e-03])
sherpa-22> sigma_s3=sqrt(array([1.1168e+01,3.2521e-05]))

sherpa-23> F4=array([[8.83838290e-01,   1.24255379e+01],[7.11544836e-02,   3.00000000e+02]])
sherpa-24> C4=array([9.,2.])
sherpa-25> ix4=array([0,1])
sherpa-26> s4=array([1.1070e+01,7.7397e-03])
sherpa-27> sigma_s4=sqrt(array([1.2351e+01,3.2228e-05]))

sherpa-28> aper1=xap.Apertures(F1)
sherpa-29> aper2=xap.Apertures(F2)
sherpa-30> aper3=xap.Apertures(F3)
sherpa-31> aper4=xap.Apertures(F4)
sherpa-32> load_user_model(aper1,"ap1")
sherpa-33> load_user_model(aper2,"ap2")
sherpa-34> load_user_model(aper3,"ap3")
sherpa-35> load_user_model(aper4,"ap4")

sherpa-39> add_user_pars("ap1",['s_1','b_1'],s1,parmins=s1-5.0*sigma_s1,parmaxs=s1+5.0*sigma_s1)
sherpa-40> add_user_pars("ap2",['s_2','b_2'],s2,parmins=s2-5.0*sigma_s2,parmaxs=s2+5.0*sigma_s2)
sherpa-41> add_user_pars("ap3",['s_3','b_3'],s3,parmins=s3-5.0*sigma_s3,parmaxs=s3+5.0*sigma_s3)
sherpa-42> add_user_pars("ap4",['s_4','b_4'],s4,parmins=s4-5.0*sigma_s4,parmaxs=s4+5.0*sigma_s4)
sherpa-43> set_model(1,ap1)
sherpa-44> set_model(2,ap2)
sherpa-45> set_model(3,ap3)
sherpa-46> set_model(4,ap4)
sherpa-47> load_arrays(1,ix1,C1,Data1D)
sherpa-48> load_arrays(2,ix2,C2,Data1D)
sherpa-49> load_arrays(3,ix3,C3,Data1D)
sherpa-50> load_arrays(4,ix4,C4,Data1D)

sherpa-51> show_model(1)
Model: 1
usermodel.ap1
   Param        Type          Value          Min          Max      Units
   -----        ----          -----          ---          ---      -----
   ap1.s_1      thawed       9.9756     -7.02955      26.9807
   ap1.b_1      thawed     0.014316   -0.0232187    0.0518507


sherpa-52> show_model(2)
Model: 2
usermodel.ap2
   Param        Type          Value          Min          Max      Units
   -----        ----          -----          ---          ---      -----
   ap2.s_2      thawed       15.599     -5.35392      36.5519
   ap2.b_2      thawed    0.0099108   -0.0230873    0.0429089


sherpa-53> show_model(3)
Model: 3
usermodel.ap3
   Param        Type          Value          Min          Max      Units
   -----        ----          -----          ---          ---      -----
   ap3.s_3      thawed       9.9643     -6.74498      26.6736
   ap3.b_3      thawed     0.007922   -0.0205916    0.0364356


sherpa-54> show_model(4)
```

```
Model: 4
usermodel.ap4
   Param        Type          Value          Min          Max     Units
   -----        ----          -----          ---          ---     -----
   ap4.s_4      thawed        11.07     -6.50199       28.642
   ap4.b_4      thawed    0.0077397   -0.0206452    0.0361246


sherpa-58> set_stat("cash")
sherpa-59> set_method("moncar")
sherpa-62> ap1.s_1=ap2.s_2
sherpa-63> ap2.s_2=ap3.s_3
sherpa-64> ap3.s_3=ap4.s_4
sherpa-65> fit(1,2,3,4)
Datasets              = 1, 2, 3, 4
Method                = moncar
Statistic             = cash
Initial fit statistic = -95.8047
Final fit statistic   = -96.3303 at function evaluation 3932
Data points           = 8
Degrees of freedom    = 3
Change in statistic   = 0.52559
   ap1.b_1        0.0107173
   ap2.b_2        0.00762721
   ap3.b_3        0.0040971
   ap4.s_4        10.6544
   ap4.b_4        0.00412601

sherpa-67> covar()
Datasets              = 1, 2, 3, 4
Confidence Method     = covariance
Iterative Fit Method  = None
Fitting Method        = moncar
Statistic             = cash
covariance 1-sigma (68.2689%) bounds:
   Param            Best-Fit  Lower Bound  Upper Bound
   -----            --------  -----------  -----------
   ap1.b_1         0.0107173  -0.00660614   0.00660614
   ap2.b_2        0.00762721   -0.0057842    0.0057842
   ap3.b_3         0.0040971  -0.00441296   0.00441296
   ap4.s_4           10.6544     -1.73312      1.73312
   ap4.b_4        0.00412601  -0.00443229   0.00443229
sherpa-68> stats,accept,params=get_draws(niter=10000)
Using Priors:
ap1.b_1: <function flat at 0x105cbc6e0>
ap2.b_2: <function flat at 0x105cbc6e0>
ap3.b_3: <function flat at 0x105cbc6e0>
ap4.s_4: <function flat at 0x105cbc6e0>
ap4.b_4: <function flat at 0x105cbc6e0>
sherpa-69> xap.write_draws("sim_1234_draws.fits",['b_1','b_2','b_3','s_4','b_4'],stats,accept,params)
sherpa-70>
```