NPTFit: A Code Package for Non-Poissonian Template Fitting

Siddharth Mishra-Sharma1, Nicholas L. Rodd2, and Benjamin R. Safdi2

1 Department of Physics, Princeton University, Princeton, NJ 08544, USA; smsharma@princeton.edu
2 Center for Theoretical Physics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA; rodd@mit.edu, bsafdi@mit.edu

Received 2016 December 20; revised 2017 April 1; accepted 2017 April 11; published 2017 May 16

Abstract

We present NPTFit, an open-source code package, written in Python and Cython, for performing non-Poissonian template fits (NPTFs). The NPTF is a recently developed statistical procedure for characterizing the contribution of unresolved point sources (PSs) to astrophysical data sets. The NPTF was first applied to Fermi gamma-ray data to provide evidence that the excess of ~GeV gamma-rays observed in the inner regions of the Milky Way likely arises from a population of sub-threshold point sources, and the NPTF has since found additional applications studying sub-threshold extragalactic sources at high Galactic latitudes. The NPTF generalizes traditional astrophysical template fits to allow for the ability to search for populations of unresolved PSs that may follow a given spatial distribution. NPTFit builds upon the framework of the fluctuation analyses developed in X-ray astronomy, thus it likely has applications beyond those demonstrated with gamma-ray data. The NPTFit package utilizes novel computational methods to perform the NPTF efficiently. The code is available at https://github.com/bsafdi/NPTFit and up-to-date and extensive documentation may be found at http://nptfit.readthedocs.io.

Key words: methods: statistical

1. Introduction

Astrophysical point sources (PSs), which are defined as sources with angular extent smaller than the resolution of the detector, play an important role in virtually every analysis utilizing images of the cosmos. It is useful to distinguish between resolved and unresolved PSs; the former may be detected individually at high significance, while members of the latter population are by definition too dim to be detected individually. However, unresolved PSs—due to their potentially large number density—can be a leading and sometimes pesky source of flux across wavelengths. Recently, a novel analysis technique called the non-Poissonian template fit (NPTF) has been developed for characterizing populations of unresolved PSs at fluxes below the detection threshold for finding individually significant sources (Lee et al. 2015, 2016). The technique expands upon the traditional fluctuation analysis technique (see, for example, (Miyaji & Griffiths 2002; Miyaji & Hogg 2011)), which analyzes the aggregate photon-count statistics of a data set to characterize the contribution from unresolved PSs, by additionally incorporating spatial information both for the distribution of unresolved PSs and for the potential sources of non-PS emission. In this work, we present a code package called NPTFit for numerically implementing the NPTF in Python and Cython.

The most up-to-date version of the open-source package NPTFit may be found at https://github.com/bsafdi/NPTFit and the latest documentation at http://nptfit.readthedocs.io. In addition, the version used in this paper has been archived at https://zenodo.org/record/380469#.WN_pSFyvMV.

The NPTF generalizes traditional astrophysical template fits. Template fitting is useful for pixelated data sets consisting of some number of photon counts $n_p$ in each pixel $p$, and it typically proceeds as follows. Given a set of model parameters $\theta$, the mean number of predicted photon counts $\mu_p(\theta)$ in the pixel $p$ may be computed. More specifically, $\mu_p(\theta) = \sum_{\ell} T^{(S)}_{p,\ell}(\theta)$, where $\ell$ is an index of the set of templates $T^{(S)}_{p,\ell}$, whose normalizations and spatial morphologies may depend on the parameters $\theta$. These templates may, for example, trace the gas-distribution or other extended structures that are expected to produce photon counts. Then, the probability to detect $n_p$ photons in the pixel $p$ is simply given by the Poisson distribution with mean $\mu_p(\theta)$. By taking a product of the probabilities over all pixels, it is straightforward to write down a likelihood function as a function of $\theta$.

The NPTF modifies this procedure by allowing for non-Poissonian photon-count statistics in the individual pixels. That is, unresolved PS populations are allowed to be distributed according to spatial templates, but in the presence of unresolved PSs the photon-count statistics in individual pixels, as parameterized by $\theta$, no longer follow Poisson distributions. This is heuristically because we now have to ask two questions in each pixel: first, what is the probability, given the model parameters $\theta$ that now also characterize the intrinsic source-count distribution of the PS population, that there are PSs within the pixel $p$, and second, given that PS population, what is the probability to observe $n_p$ photons?

It is important to distinguish between resolved and unresolved PSs. Once a PS is resolved—that is, once its location and flux is known—that PS may be accounted for by its own Poissonian template. Unresolved PSs are different because their locations and fluxes are unknown. When we characterize unresolved PSs with the NPTF, we characterize the entire population of unresolved sources, following a given spatial distribution, based on how that population modifies the photon-count statistics.

The NPTF has played an important role recently in addressing various problems in gamma-ray astroparticle physics with data collected by the Fermi-LAT gamma-ray telescope. The NPTF was developed to address the excess of gamma-rays observed by Fermi at ~GeV energies originating from the inner regions of the Milky Way (Goodenough & Hooper 2009; Boyarsky et al. 2011; Hooper & Goodenough

8 http://fermi.gsfc.nasa.gov/
The NPTF originates from the older fluctuation analysis technique, which is sometimes referred to as the P(D) analysis. This technique has been used extensively to study the flux of unresolved X-ray sources (Georganopoulos et al. 1993; Hasinger et al. 1993; Gendreau et al. 1998; Perri & Giommi 2000; Miyaji & Griffiths 2002). In these early works, the photon-count probability distribution function (PDF) was computed numerically for different PS source-count distributions using Monte Carlo (MC) techniques. The fluctuation analysis was first applied to gamma-ray data in (Malyshew & Hogg 2011),\(^5\) and in that work the authors developed a semi-analytic technique utilizing probability generating functions for calculating the photon-count PDF. The code package NPTFit presented in this work uses this formalism for efficiently calculating the photon-count PDF. The specific form of the likelihood function for the NPTF, while reviewed in this work, was first presented in (Lee et al. 2016). The works (Lee et al. 2016; Linden et al. 2016; Lisanti et al. 2016) utilized an early version of NPTFit to perform their numerical analyses.

The NPTFit code package has a python interface, though the likelihood evaluation is efficiently implemented in cython (Behnel et al. 2011). The user-friendly interface allows for an arbitrary number of PS and smooth templates. The PS templates are characterized by pixel-dependent source-count distributions \(dN_p/dF = T_p^{(PS)}dN/dF\), where \(T_p^{(PS)}\) is the spatial template tracking the distribution of point sources on the sky and \(dN/dF\) is the pixel-independent source-count distribution. The distribution \(dN_p/dF\) quantifies the number of sources \(dN_p\) that contributes flux between \(F\) and \(F + dF\) in the pixel \(p\). The \(dN/dF\) are parameterized as multiply broken power laws, with an arbitrary number of breaks. The code is able to account for both an arbitrary exposure map (accounting for the pointing strategy of an instrument) as well as an arbitrary point-spread function (PSF, accounting for the instrument’s finite angular resolution) in translating between flux \(F\) and counts \(S\).

NPTFit has a built-in interface with MultiNest (Feroz et al. 2009; Buchner et al. 2014), which efficiently implements nested sampling of the posterior distribution and Bayesian evidence for the user-specified model, given the specified data and instrument response function, in the Bayesian framework (Skilling 2006; Feroz & Hobson 2008; Feroz et al. 2013). The interface handles the Message Passing Interface, so that inference may be performed efficiently using parallel computing. A basic analysis package is provided to facilitate the easy extraction of the most relevant data from the posterior distribution and quick plotting of the MultiNest output. The preferred format of the data for NPTFit is HEALPix (Gorski et al. 2005) (a nested equal-area pixilation scheme of the sky), although the the code is also able to handle non-HEALPix data arrays. Note that the code package may also be used to simply extract the NPTF likelihood function so that NPTFit may be interfaced with any numerical package for Bayesian or frequentist inference.

A large set of example Jupyter (Pérez & Granger 2007) notebooks and Python files are provided to illustrate the code. The examples utilize 413 weeks of processed Fermi Pass 8 data in the UltradeepN veto event class collected between 2008 August 4 and 2016 July 7 in the energy range from 2 to 20 GeV. We restrict this data set to the top quartile as graded by PSF reconstruction and further apply the standard quality cuts.

---

\(^{5}\) The fluctuation analysis has more recently been applied to both gamma-ray (Feyereisen et al. 2015) and neutrino (Feyereisen et al. 2016) data sets.
DATA_QUAL==1 & LAT_CONFIG==1, as well as restricting the zenith angle to be less than 90°. This data is made available in the code release. Moreover, the example notebooks illustrate many of the main results in (Lee et al. 2016; Linden et al. 2016; Lisanti et al. 2016).

In addition to the above, the base NPTFit code makes use of the Python packages corner (Foreman-Mackey et al. 2016), matplotlib (Hunter 2007), mpmath (Johansson et al. 2013), GSL (Galassi et al. 2015) and numpy (Oliphant 2006).

The rest of this paper is organized as follows. Section 2 outlines the framework of the NPTF in more detail. Section 3 highlights the key classes and features in the NPTFit code package and usage instructions. In Section 4, we present an example of how to perform an NPTF scan using NPTFit, looking at the Galactic Center with Fermi data to reproduce aspects of the main results of (Lee et al. 2016). We conclude in Section 5. Appendices A–C describe further details behind the mathematical framework of the NPTF.

2. The Non-Poissonian Template Fit

This section reviews the NPTF, which was first presented in Lee et al. (2016) and is described in more detail in Linden et al. (2016) and Lisanti et al. (2016) (see also Malyshev & Hogg 2011; Lee et al. 2015; Zechlin et al. 2016a, 2016b). The NPTF is used to fit a model $\mathcal{M}$ with parameters $\theta$ to a data set $d$ consisting of counts $n_p$ in each pixel $p$. The likelihood function for the NPTF is then simply

$$p(d|\theta, \mathcal{M}) = \prod_p p_{n_p}^{(p)}(\theta),$$

where $p_{n_p}^{(p)}(\theta)$ gives the probability of drawing $n_p$ counts in the given pixel $p$, as a function of the parameters $\theta$. The main computational challenge, of course, is in computing these probabilities.

It is useful to divide the model parameters into two different categories: the first category describes smooth templates, while the second category describes PS templates. We describe each category in turn, starting with the smooth templates.

For most applications, the data has the interpretation of being a two-dimensional (2D) pixelated map consisting of an integer number of counts in each pixel. The smooth templates may be used to predict the mean number of counts $\mu_p(\theta)$ in each pixel $p$

$$\mu_p(\theta) = \sum_\ell \mu_{p,\ell}(\theta).$$

Above, $\ell$ is an index over templates and $\mu_{p,\ell}(\theta)$ denotes the mean contribution of the $\ell$th template to pixel $p$ for parameters $\theta$. In principle, $\theta$ may describe both the spatial morphology as well as the normalization of the templates. However, in the current implementation of the code, the Poissonian model parameters simply characterize the overall normalization of the templates: $\mu_{p,\ell}(\theta) = A_{p,\ell}(\theta)T^{(S)}_{p,\ell}$. Here, $A_{p,\ell}$ is the normalization parameter and $T^{(S)}_{p,\ell}$ is the $\ell$th template, which takes values over all pixels $p$ and is independent of the model parameters. The superscript (S) implies that the template is a counts templates, which is to be contrasted with a flux template, for which we use the symbol ($F$). The two are related by the exposure map of the instrument $E_p$: $T^{(S)}_p = E_pT^{(F)}_p$. In the case where we only have smooth, Poissonian templates, the probabilities are then given by the Poisson distribution

$$p_{n_p}^{(p)}(\theta) = \frac{\mu_{n_p}^{n_p}(\theta)}{n_p!}e^{-\mu_{n_p}(\theta)}.$$  

In the presence of unresolved PS templates, the probabilities $p_{n_p}^{(p)}(\theta)$ are no longer Poissonian functions of the model parameters $\theta$. Each PS template is characterized by a pixel-dependent source-count distribution $dN_p/dF$, which describes the differential number of sources per pixel per unit flux interval. In this work, we model the source-count distribution by a multiply broken power law:

$$\frac{dN_p}{dF}(F; \theta) = A(\theta)T^{(PS)}_p.$$

Above, we have parameterized the source-count distribution with an arbitrary number of breaks $k$, denoted by $F_{b,i}$ with $i \in [1, 2, \ldots, k]$, and $k + 1$ indices $n_i$ with $i \in [1, 2, \ldots, k + 1]$. The spatial dependence of the source-count distribution is accounted for by the overall factor $A(\theta)T^{(PS)}_p$, where $A(\theta)$ is the pixel-independent normalization, which is a function of the model parameters, and $T^{(PS)}_p$ is a template describing the spatial distribution of the PSs. More precisely, the
number of sources $N_{PS} = \int dF dN_p / dF$ (and the total PS flux $F_{PS} = \int dF dN_p / dF$) in pixel $p$, for a fixed set of model parameters $\theta$, follows the template $T_P^{(PS)}$. On the other hand, the locations of the flux breaks and the indices are taken to be fixed between pixels.\(^6\)

To summarize, a PS template described by a broken power law with $k$ breaks has $2(k + 1)$ model parameters describing the locations of the breaks, the power-law indices, and the overall normalization. For example, if we take a single break, then the PS model parameters may be denoted as $\{A, F_b, n_1, n_2\}$. Additionally, a spatial template $T_S^{(PS)}$ must be specified, which describes the distribution of the number of sources (and total flux) with pixel $p$.

Notice that when we discussed the Poissonian templates, we used the counts templates $T^{(S)}$ and talked directly in terms of counts $S$, while so far in our discussion of the unresolved PS templates we used the point source distribution template $T^{(PS)}$ and wrote the source-count distribution $dN / dF$ in terms of flux $F$. Of course, as the total flux from a distribution of point sources is also proportional to the template $T^{(PS)}$, it can be thought of as a flux template, however conceptually it is being used to track the distribution of the sources rather than the flux they produce. For this reason, we have chosen to distinguish the two. Moreover, in the presence of a non-trivial PSF, $T^{(S)}$ should also be smoothed by the PSF to account for the instrument response function. That is, $T^{(S)}$ is a template for the observed counts taking into account the details of the instrument, while $T^{(PS)}$ ($T^{(F)}$) is a map of the physical point sources (flux), which is independent of the instrument. In photon-counting applications, the exposure map $E_p$ often has units of cm$^2$ s and flux has units of counts cm$^{-2}$ s$^{-1}$.

For the unresolved PS templates, we also need to convert the source-count distribution from flux to counts. This is done by a simple change of variables:

$$
\frac{dN_p}{dS}(S; \theta) = \frac{1}{E_p} \frac{dN_p}{dF}(F = S/E_p; \theta),
$$

which implies that for a non-Poissonian template the spatial dependence of $dN_p / dS$ is given by $T_P^{(PS)} / E_p$. This inverse exposure scaling may seem surprising, but it is straightforward to confirm that the mean number of counts in a given pixel, $\int dS dN_p / dS$, is given by $E_p T_P^{(PS)}$, as expected, up to pixel-independent factors.

As an important aside, the template $T^{(S)}$ used by the Poissonian models needs to be smoothed by the PSF. Incorporating the PSF into the unresolved PS models, on the other hand, is more complicated and is not accomplished simply by smoothing the spatial template. Indeed, $T^{(PS)}$ should remain unsmoothed by the PSF when used for non-Poissonian scans.

In the remainder of this section, we briefly overview the mathematical framework behind the computation of the $T_P^{(PS)}(\theta)$ with NPTFit; however, details of the algorithms used to calculate these probabilities in practice, along with more in-depth explanations, are given in Appendices A–C. We use the probability generating function formalism, following (Malyshchev & Hogg 2011), to calculate the probabilities. For a discrete probability distribution $p_k$, with $k = 0, 1, 2, \ldots$, the generating function is defined as

$$
P(t) = \sum_{k=0}^{\infty} p_k t^k,
$$

from which we can recover the probabilities

$$
p_k = \frac{d^k P(t)}{dt^k} \bigg|_{t=0}.
$$

The key feature of generating functions exploited here is that the generating function of a sum of two independent random variables is simply the product of the individual generating functions.

The probability generating function for the smooth templates, as a function of $\theta$, is simply given by

$$
P_P(t; \theta) = \prod_p \exp[\mu_p(\theta)(t - 1)].
$$

The probability generating function for an unresolved PS template, on the other hand, takes a more complicated form:

$$
P_{NP}(t; \theta) = \prod_p \exp \left[ \sum_{m=1}^{\infty} x_{p,m}(\theta)(t^m - 1) \right],
$$

where

$$
x_{p,m}(\theta) = \int_0^\infty dS dN_p(S; \theta) \int_0^1 d\rho(f) (\frac{FS}{m!})^m e^{-FS}.
$$

Above, $\rho(f)$ is a function that takes into account the PSF, which we describe in more detail in Appendix A. In the presence of a non-trivial PSF, the flux from a single source is smeared among pixels. The distribution of flux fractions among pixels is described by the function $\rho(f)$, where $f$ is the flux fraction. By definition, $\rho(f) df$ equals the number of pixels which, on average, contain between $f$ and $f + df$ of the flux from a PS; the distribution is normalized such that $\int_0^1 df \rho(f) = 1$. If the PSF is a $\delta$-function, then $\rho(f) = \delta(f - 1)$.

Putting aside the PSF correction for the moment, the $x_{p,m}$ have the interpretation of being the average number of $m$-count PSs within the pixel $p$, given the distribution $dN_p(S; \theta) / dS$. The generating function for $x_{m}$, $m$-count sources is simply $e^{\mu_m (f - 1)}$ (see (Malyshchev & Hogg 2011) or Appendix A), which then leads directly to (9). The PSF correction, through the distribution $\rho(f)$, incorporates the fact that PSs only contribute some fraction of their flux within a given pixel.

### 3. NPTFit: Orientation

NPTFit implements the NPTF, as described above, in Python. In this section, we give a brief orientation to the code package and its main classes. A more thorough description of the code and its uses is available in the online documentation.\(^7\)

```python
class NPTFit.nptfit.NPTF
```

This is the main class used to set up and perform non-Poissonian and Poissonian template scans. It is initialized by

\(^6\) In principle, the breaks and indices could also vary between pixels. However, in the current version of NPTFit, only the number of sources (and, accordingly, the total flux) is allowed to vary between pixels.

\(^7\) http://nptfit.readthedocs.io
with keywords

| Argument    | Default   | Purpose             | Type |
|-------------|-----------|---------------------|------|
| tag         | 'Unmasked' | Label of scan       | str  |
| work_dir    | None      | Output directory    | str  |

If no work_dir is specified, the code will default to the current directory. This is the directory where all output is stored. Specifying a tag will create an additional folder with that name within the work_dir for the output.

The data, exposure map, and templates are loaded into the nptfit.NPTF instance after initialization (see the example in Section 4). The data and exposure map are loaded by

```python
nptf.load_data(data, exposure)
```

Here, 'data' and 'exposure' are one-dimensional (1D) numpy arrays. The recommended format for these arrays is the HEALPix format, so that all pixels are equal area, although the code is able to handle arbitrary data and exposure arrays so long as they are of the same length. The templates are added by

```python
nptf.add_template(template, key, units='counts')
```

Here, 'template' is a 1D numpy array of the same length as the data and exposure map. 'key' is a string that will be used to refer to the template later on, and 'units' specifies whether the template is a counts template (keyword 'counts') or a flux template (keyword 'flux') in units counts cm$^{-2}$ s$^{-1}$. The default, if unspecified, is units = 'counts'. The template should be pre-smoothed by the PSF if it is going to be used for a Poissonian model. If the template is going to be used for a non-Poissonian model, either choice for units is acceptable, though in the case of 'counts' the template should simply be the product of the exposure map times the flux template and not smoothed by the PSF.

The user also has the option of loading in a mask that reduces the region of interest (ROI) to a subset of the pixels in the data, exposure, and template arrays. This is done through the command

```python
nptf.load_mask(mask)
```

where mask is a boolean numpy array of the same length as the data and exposure arrays. Pixels in mask should be either True or False; by convention, pixels that are True will be masked, while those that are False will not be masked. Note that if performing an analysis with non-Poissonian templates, regions where the exposure map is identically zero should be explicitly masked.

Afterward, Poissonian and non-Poissonian models may be added to the instance using the available templates. An arbitrary number of Poissonian and non-Poissonian models may be added to the scan. Moreover, each non-Poissonian model may be specified in terms of a multiply broken power law with a user-specified number of breaks, as in (4).

Poissonian models are added sequentially using the syntax

```python
nptf.add_pois_model(template_name, model_tag, prior_range=[], log_prior=True, fixed_params=None, units='counts')
```

Any of the model parameters may be fixed to a user-specified value instead of floated in the scan. For those parameters that are floated in the scan, a prior range needs to be specified along with whether the prior is flat or log-flat. Note that if log_prior = True, then the prior range is set with respect to log$_{10}$ of the linear prior range. For example, if we want to scan the normalization of a template over the range from [0.1, 10] with a log-flat prior, then we would set log_prior = True and prior_range = [-1, 1]. In this case, it might make sense to label the model with $\mathcal{M} = 'S\log_{10}A'$ to emphasize that the actual model parameter is the log of the normalization; this label will appear in various plots made using the provided analysis class for visualizing the posterior.

The non-Poissonian models are added with a similar syntax:

```python
nptf.add_non_pois_model(template_name, model_tag, prior_range=[], log_prior=True, dndx_model='specify_breaks', fixed_params=None, units='counts')
```

The template_name keyword is the same as for the Poissonian models. The rest of the keywords are

| Argument   | Default          | Purpose            | Type          |
|------------|------------------|--------------------|---------------|
| model_tag  | -                | LATEX-ready label  | [str, str, ...] |
| prior_range| []               | Prior [min, max]   | [float, float] |
| log_prior  | False            | Log/linear-flat prior | [bool, bool, ...] |
| fixed     | False            | Is template fixed | bool |
| fixed_norm| 1.0              | Norm if fixed      | float |

The syntax for adding non-Poissonian models is that the model parameters are specified by $[A, n_1, n_2, \ldots, n_{k-1}, S_{0,1}, S_{0,2}, \ldots, S_{0,k}]$ for a broken power law with $k$ breaks. As such, the model_tag, prior_range, and log_prior are now arrays where each entry refers to the respective model parameter. The code automatically determines the number of breaks by the length

---

Mishra-Sharma, Rodd, & Safdi

(Continued)

False, fixed=False, fixed_norm=1.0)
of the model_tag array. The arrays prior_range and log_prior should only include entries for model parameters that will be floated in the scan. Any model parameter may be fixed using the fixed_params array, with the syntax such that fixed_params = [[i, c_i], [j, c_j]] would fix the i\textsuperscript{th} model parameter to c\textsubscript{i} and the j\textsuperscript{th} to c\textsubscript{j}, where the parameter indexing starts from 0.

The units keyword determines whether the priors for the breaks in the source-count distribution (and also the fixed parameters, if any are given) will be specified in terms of 'flux' or 'counts'. The relation between flux and counts varies between pixels if the exposure map is non-trivial. For this reason, it is more appropriate to think of the breaks in the source-count distribution in terms of flux. The keyword 'counts' still specifies the breaks in the source-count distribution in terms of flux, with the relation between counts and flux given through the mean of the exposure map mean(E): \( F_{b,j} = S_{b,j} / \text{mean}(E) \).

The dnds_model keyword has the options 'specify_breaks' and 'specify_relative_breaks'. If 'specify_breaks' is chosen, which is the default, then the breaks are the model parameters. If instead 'specify_relative_breaks' is chosen, the full set of model parameters is given by \([A, n_1, n_2, \ldots, \lambda_1, \ldots, \lambda_k]\). Here, \( S_{b,0} \) is the highest break and the lower breaks are determined by \( S_{b,0} = \lambda_j S_{b,0-1} \). Note that the prior ranges for the \( \lambda_i \)'s should be between 0 and 1 (for linear flat), since \( S_{b,i} < S_{b,i-1} \).

After setting up a scan, the configuration is finished by executing the command

```python
nptf.configure_for_scan(f_ary=1.0, df_rho_div_f_ary=1.0, nexp=1)
```

For a purely Poissonian scan, none of the keywords above need to be specified. For non-Poissonian scans, f_ary and df_rho_div_f_ary incorporate the PSF correction. In particular, f_ary is a discretized list of \( f \) values between 0 and 1, while df_rho_div_f_ary is a discretized list of \( d\rho(f)/f \) at those \( f \) values. A class is provided for computing these lists; it is described later in this section. If no keywords are given for these two arrays they default to the case of a \( \delta \) function PSF.

The keyword nexp, which defaults to 1, is related to the exposure correction in the calculation of the source-count distribution \( dN_p/dS \) from \( dN_p/dF \). In many applications, it is computationally too expensive to perform the mapping in (5) in each pixel. The overall pixel-dependent normalization factor \( T^\text{PSF}_p/E_p \) factorizes from many of the internal computations, and as a result this contribution to the exposure correction is performed in every pixel. However, it is useful to perform the mapping from flux to counts, which should be performed uniquely in each pixel \( F = S/E_p \), using the mean exposure within small sub-regions. Within a given sub-region, we map flux to counts using \( F = S/\text{mean}(E) \), where the mean is taken over all pixels in the sub-region. The number of sub-regions is given by nexp, and all sub-regions have approximately the same area. As nexp approaches the number of pixels, the approximation becomes exact; however, for many applications the approximation converges for a relatively small number of exposure regions. We recommend verifying, in any application, that results are stable as nexp is increased.

After configuring the NPTF instance, the log-likelihood may be extracted, as a function of the model parameters, in addition to the prior range. The log-likelihood and prior range may then be used with any external package for performing Bayesian or frequentist inference. This is particularly useful if the user would like to combine likelihood functions between different energy bins or otherwise add to the default likelihood function, for example, incorporating nuisance parameters beyond those associated with individual templates. The package MultiNest, however, is already incorporated into the NPTF class and may be run immediately after configuring the NPTF instance. This is done simply by executing the command

```python
nptf.perform_scan(run_tag=None, nlive=100)
```

where nlive is an integer that specifies the number of live points used in the sampling of the posterior distribution. MultiNest recommends an nlive \( \sim \) 500–1000, though the parameter defaults to 100 if unspecified for quick test runs. Additional MultiNest arguments may be passed as a dictionary through the optional pymultinest_options keyword (see the online documentation for more details). The optional keyword run_tag is used to create a sub-folder for the MultiNest output with that name.

After a scan has been run (or if a scan has been run previously and saved), the results may be loaded through the command

```python
nptf.load_scan(run_tag=None)
```

The MultiNest chains, which give a discretized view of the posterior distribution, may then be accessed through, for example, nptf.samples. An instance of the PyMultiNest analyzer class may be accessed through nptf.a. A small analysis package, described later in this section, is also provided for performing a few common analyses.

```python
class NPTFit.psf_correction.PSFCorrection
```

This is the class used to construct the arrays f_ary and df_rho_div_f_ary for the PSF correction. An instance of PSFCorrection is initialized through

```python
pc_inst = PSFCorrection.PSFCorrection(
    psf_dir=psf_dir=None, num_f_bins=10, n_psf=50000, n_pts_per_psf=1000, f_trunc=0.01, nsided=128, psf_sigma_deg=None,
    delay_compute=False
)
```

with keywords

| Argument         | Default | Purpose                                      | Type  |
|------------------|---------|----------------------------------------------|-------|
| psf_dir          | None    | Where PSF arrays are stored                   | str   |
| num_f_bins       | 10      | Number of linear-spaced points                | int   |
| n_psf            | 50000   | Number of MC simulations for determining      | int   |
| n_pts_per_psf    | 1000    | Number of points drawn for each MC simulation | int   |
| f_trunc          | 0.01    | Minimum \( f \) value                        | float |
| nsided           | 128     | HEALPix parameter for size of map            | int   |
| psf_sigma_deg    | None    | Standard deviation \( \sigma \) of 2D Gaussian PSF | float |
Note that the arrays \( f_{\text{ary}} \) and \( \text{df}_r \text{ho}_\text{div}_f_{\text{ary}} \) depend both on the PSF of the detector as well as the pixelation of the data; at present the \text{PSFCorrection} class requires the pixelation to be in the HEALPix pixelation.

The keyword \text{psf}_\text{dir} points to the directory where the \( f_{\text{ary}} \) and \( \text{df}_r \text{ho}_\text{div}_f_{\text{ary}} \) will be stored; if unspecified, they will be stored to the current directory. The \( f_{\text{ary}} \) consists of \text{num}_f_{\text{bins}} entries linear spaced between 0 and 1. The PSF correction involves placing many \( (\text{nsf}) \) PSFs at random positions on the HEALPix map, drawing \( \text{n}_\text{pts}_\text{per}_\text{psf} \) points from each PSF, and then looking at the distribution of points among pixels. The larger \( \text{n}_\text{psf} \) and \( \text{n}_\text{pts}_\text{per}_\text{psf} \), the more accurate the computation of \( \text{df}_r \text{ho}_\text{div}_f_{\text{ary}} \) will be. However, the computation time of the PSF arrays also increases as these parameters are increased.

By default the \text{PSFCorrection} class assumes that the PSF is a 2D Gaussian distribution:

\[
\text{PSF}(r) = \frac{1}{2\pi \sigma^2} \exp\left[-\frac{r^2}{2 \sigma^2}\right]
\]

Here, \( \text{PSF}(r) \) describes the spread of arriving counts with angular distance \( r \) away from the arrival direction. The parameter \text{psf}_\text{sigma}_\text{deg} denotes \( \sigma \) in degrees. Upon initializing \text{PSFCorrection} with \text{psf}_\text{sigma}_\text{deg} specified, the class automatically computes the array \( \text{df}_r \text{ho}_\text{div}_f_{\text{ary}} \) and stores it in the \text{psf}_\text{dir} with a unique name related to the keywords. If such a file already exists in the \text{psf}_\text{dir}, then the code will simply load this file instead of recomputing it. After initialization, the relevant arrays may be accessed by \text{pc}_\text{inst}_f_{\text{ary}} \text{and pc}_\text{inst}_f_{\text{ary}} \text{div}_f_{\text{ary}}.

The \text{PSFCorrection} class can also handle arbitrary PSF functions. In this case, the class should be initialized with \text{delay}_\text{compute}=True. Then, the user should manually set the function \text{pc}_\text{inst}_f_{\text{ary}} \text{r}_\text{func} to the desired function \( \text{PSF}(r) \). This function will be discretized with \text{pc}_\text{inst}_f_{\text{ary}} \text{psf}_\text{samples} points out to \text{pc}_\text{inst}_f_{\text{ary}} \text{sample}_\text{psf}_\text{max} degrees from \( r=0 \). These two quantities also need to be manually specified. The user also needs to set \text{pc}_\text{inst}_f_{\text{ary}} \text{psf}_\text{tag} to a string that will be used for saving the PSF arrays. After these four attributes have been set manually by the user, the PSF arrays are computed and stored by executing \text{pc}_\text{inst}_f_{\text{ary}} \text{make}_\text{or}_\text{load}_\text{psf}_\text{corr}() in the \text{pc}_\text{inst}.

**Example:**

```python
def NPTFit.create_mask.make_mask_total

This function is used to make masks that can then be used to reduce the data and templates to a smaller ROI when performing the scan. While these masks can always be made by hand, this function provides a simple masking interface for maps in the HEALPix format. The \text{make_mask_total} function can mask pixels by latitude, longitude, and radius from any point on the sphere. See the online documentation (see footnote 7) for more specific examples.

**Example:**

```python

class NPTFit.dnds_analysis.Analysis

The analysis class may be used to extract useful information from the results of an NPTF performed using MultiNest. The class also has built-in plotting features for making many of the most common types of visualizations for the parameter posterior distribution. An instance of the analysis class can be instantiated by

```python

an=Analysis(nptf, mask=None, pixarea=0.)

```python

where \text{nptf} is itself an instance of the NPTF class that already has the results of a scan loaded. The keyword arguments \text{mask} and \text{pixarea} are optional. The user should specify a \text{mask} if the desired ROI for the analysis is different that that used in the scan. The user should specify a \text{pixarea} if the data is not in the HEALPix format. The code will still assume the pixels are equal area with area \text{pixarea}, which should be specified in sr.

After initialization, the intensities of Poissonian and non-Poissonian templates, respectively, may be extracted from the analysis class by the commands

```python

an.return_intensity_arrays_poisson(comp)

and

an.return_intensity_arrays_non_poisson(comp)

```python

Here, \text{comp} refers to the template key used by the Poissonian or non-Poissonian model. The arrays returned give the mean intensities of that model in the ROI in units of counts cm\(^{-2}\) s\(^{-1}\), assuming the exposure map was in units of cm\(^{2}\).s. The arrays computed over the full set of entries in the discretized posterior distribution output by MultiNest. Thus, these intensity arrays may be interpreted as the 1D posteriors for the intensities. For additional keywords that may be used to customize the computation of the intensity arrays, see the online documentation (see footnote 7).

The source-count distributions may also be accessed from the analysis class. Executing

```python

an.return_dndf_arrays(comp, flux)

```python

will return the discretized 1D posterior distribution for mean\(\text{RoD}_{\text{fit}}(F)/dF\) at flux \(F\) for the PS model with template key \text{comp}. Note that the mean is computed over pixels \(p\) in the ROI.

The 1D posterior distributions for the individual model parameters may be accessed by

```python

A\_poiss\_post = an.

return_poission_parameter_posteriors(
    comp)

```python

for Poissonian models, and

```python

A\_non\_poiss\_post, n\_non\_poiss\_post,

Sb\_non\_poiss\_post = an.

return_non_poisson_parameter_posteriors(
    comp)

```python

for non-Poissonian models. Here, \text{A\_poiss\_post} is a 1D array of the discretized posterior distribution for the Poissonian template normalization parameter. Similarly, \text{A\_non\_poiss\_post} is the posterior array for the non-Poissonian normalization parameter.
The arrays \( n_{\text{non-poiss}} \) and \( S_b_{\text{non-poiss}} \) are 2D, where—for example—\( n_{\text{non-poiss}} = [n_1, \ldots, n_2, \ldots] \) and \( n_1 \) is a 1D array for the posterior for \( n_1 \).

Another useful piece of information that may be extracted from the scan is the Bayesian evidence:

\[
l_{\text{BE}}, l_{\text{BE}}\text{err} = \text{an.get}\_\text{log}\_\text{evidence}()
\]

returns the log of the Bayesian evidence along with the uncertainty on this estimate based on the resolution of the MCMC.

For information on the plotting capabilities in the analysis class, see the online documentation (see footnote 7) or the example in the following section.

4. NPTFit: An Example

In this section, we give an example for how to perform an NPTF using NPTFit. Many more examples are available in the online documentation (see footnote 7). This particular example reproduces aspects of the main results of (Lee et al. 2016), which found evidence for a spherical population of unresolved gamma-ray PSs around the Galactic Center. The example uses the processed, public Fermi data made available with the release of the NPTFit package. The data set consists of 413 weeks of Fermi Pass 8 data in the UltracleanVeto event class (top quartile of events as ranked by PSF) from 2 to 20 GeV. The map is binned in HEALPix with \( n_{\text{side}} = 128 \).

The data, along with the exposure map and background templates, may be downloaded from \url{http://hdl.handle.net/1721.1/105492}. In the example, we will perform an NPTF on the sub-region where we mask the Galactic plane at latitude \( |b| < 2^\circ \) and mask pixels with angular distance greater than 30\(^\circ\) from the Galactic Center. We also mask identified PSs in the 3FGL PS catalog (Acero et al. 2015) at 95\% containment using the provided PS mask, which is added to the geometric mask. We include smooth templates for diffuse gamma-ray emission in the Milky Way (using the Fermi p6v11 diffuse model), isotropic emission (which can also absorb instrumental backgrounds), and emission following the Fermi bubbles, which are taken to be uniform in flux following the spatial template in (Su et al. 2010). We also include a DM template, which traces the line of sight integral of the square of a canonical NFW density profile.

We additionally include point source (non-Poissonian) models for the DM template, as well as for a disk template which corresponds to a doubly exponential thin-disk source distribution with scale height 0.3 kpc and radius 5 kpc. The source-count distributions for these are parameterized by singly broken power laws, each described by four parameters \( \{A, \phi, b, n_1, n_2\} \).

4.1. Setting Up the Scan

We begin the example by loading in the relevant modules, described in the previous section, that we will need to set up, perform, and analyze the scan.

```python
import numpy as np
# module for performing scan
from NPTFit import nptfit
# module for creating the mask
from NPTFit import create_mask as cm
# module for determining the PSF correction
```

Note that templates \( gce_{\text{np}} \) and \( dsk_{\text{np}} \) intended to be used in non-Poissonian models should trace the underlying PS distribution, without exposure correction, and are added with the keyword units='PS'.

Next, we create an instance of the NPTF class, which is used to configure and perform a scan.

```python
n = nptfit.NPTF(tag='GCE_Example')
```

We assume here that the supplementary Fermi data has been downloaded to a directory ‘fermi_data’. Then we may load in the data and exposure maps by

```python
fermi_data = np.load('fermi_data/fermidata_counts.npy').astype(int)
fermi_exposure = np.load('fermi_data/fermidata_exposure.npy')
```

Importantly, note that the exposure map has units of cm\(^2\)s. Next, we use the create_mask class to generate our ROI mask, which consists of both the geometric mask and the PS mask loaded in from the ‘fermi_data’ directory:

```python
pscmask = np.array(np.load('fermi_data/fermidata_pscmask.npy'), dtype=bool)
mask = cm.make_mask_total(band_mask=True, band_mask_range=2, mask_ring=True, inner=0, outer=30,
                          custom_mask=pscmask)
```

The templates may also be loaded in from this directory,

```python
dif = np.load('fermi_data/template_dif.npy')
iso = np.load('fermi_data/template_iso.npy')
bub = np.load('fermi_data/template_bub.npy')
gce = np.load('fermi_data/template_gce.npy')
dsk = np.load('fermi_data/template_dsk.npy')
```

These templates are counts map (i.e., flux maps times the exposure map) that have been pre-smoothed by the PSF (except for the disk-correlated template labeled dsk). We then add them to our NPTF instance with appropriately chosen keywords:

```python
n.add_template(dif, 'dif')
n.add_template(iso, 'iso')
n.add_template(bub, 'bub')
n.add_template(gce, 'gce')
n.add_template(dsk, 'dsk')
```

# remove the exposure correction for PS templates
rescale = fermi_exposure.mean()
```

Note that templates \( gce_{\text{np}} \) and \( dsk_{\text{np}} \) intended to be used in non-Poissonian models should trace the underlying PS distribution, without exposure correction, and are added with the keyword units='PS'.

(Continued)
4.2. Adding Models

Now that we have loaded in all of the external data and templates, we can add models to our NPTF instance. First, we add in the Poissonian models:

```python
n.add_poisson_model('diff', '$A_\text{\math{diff}}' \math{diff} $', False, fixed=True, fixed_norm=-14.67)
n.add_poisson_model('iso', '$A_\text{\math{iso}}$ $\math{iso}$ $S'$, [0, 2], False)
n.add_poisson_model('gce', '$A_\text{\math{gce}}$ $\math{gce}$ $S'$, [0, 2], False) n.add_poisson_model('bub', '$A_\text{\math{bub}}$ $\math{bub}$ $S'$, [0, 2], False)
```

All Poissonian models are taken to have linear priors, with prior ranges for the normalizations between 0 and 2. However, the normalization of the diffuse background has been fixed to the value 14.67, which is approximately the correct normalization in these units for this template, to provide an example of this syntax. Next, we add in the two non-Poissonian models:

```python
n.add_non_poisson_model('gce_np', '$A_\text{\math{gce}}$ $\math{gce}$ $S'$, [0, 1], False, $S_{0.1}^2$ $\math{gce}$ $S'$, [-6, 1], [2.05, 30], [-2, 1.95], [0.05, 40], $S_{b1}^2$ $\math{gce}$ $S'$, [0, 2], False, $S_{b2}^2$ $\math{gce}$ $S'$, [0, 2], False)
n.add_non_poisson_model('dsk_np', '$A_\text{\math{dsk}}$ $\math{dsk}$ $S'$, [0, 1], False, $S_{0.1}^2$ $\math{dsk}$ $S'$, [-6, 1], [2.05, 30], [-2, 1.95], [0.05, 40], $S_{b1}^2$ $\math{dsk}$ $S'$, [0, 2], False, $S_{b2}^2$ $\math{dsk}$ $S'$, [0, 2], False)
```

We have added in the models for disk-correlated and NFW-correlated (line of sight integral of the the NFW distribution squared) unresolved PS templates. Each of these models takes singly broken power law source-count distributions. In each case, the normalization parameter is taken to have a log-flat prior while the indices and breaks are taken to have linear priors. The units of the breaks are specified in terms of counts.

4.3. Configure Scan with PSF Correction

In this energy range and with this data set, the PSF may be modeled by a 2D Gaussian distribution with $\sigma = 0.1812$. From this, we are able to construct the PSF-correction arrays:

```python
pc_inst = pc.PSFCorrection(psf_sigma_deg=0.1812)
f_ary, df_rho_div_f_ary = pc_inst.f_ary, pc_inst.df_rho_div_f_ary
```

These arrays are then passed into the NPTF instance when we configure the scan:

```python
n.configure_for_scan(f_ary, df_rho_div_f_ary, nexp=1)
```

Note that since our ROI is relatively small and the exposure map does not change significantly over the region, we have a single exposure region with $n_{\text{exp}}=1$.9

4.4. Performing the Scan with MultiNest

We perform the scan using MultiNest with $n_{\text{live}}=100$ as an example to demonstrate the basic features and conclusions of this analysis while being able to perform the scan in a reasonable amount of time on a single processor, although ideally $n_{\text{live}}$ should be set to a higher value for more reliable results:

```python
n.perform_scan(nlive=100)
```

4.5. Analyzing the Results

Now we are ready to analyze the results of the scan. First, we load in relevant modules:

```python
import corner
import matplotlib.pyplot as plt
```

and then we load in the results of the scan (configured as above),

```python
n.load_scan()
```

The chains, giving a discretized view of the posterior distribution, may be accessed simply through the attribute `n.samples`. However, we will analyze the results by using the analysis class provided with NPTF. It. We make an instance of this class simply by

```python
an = nds_analysis.Analysis(n)
```

4.5.1. Make Triangle Plots

Triangle plots are a simple and quick way of visualizing correlations in the posterior distribution. Such plots may be generated through the command

```python
an.make_triangle()
```

which leads to the plot in Figure 1.

4.5.2. Plot Source-count Distributions

The source-count distributions for NFW- and disk-correlated point source models may be plotted with

```python
an.plot_source_count_median('dsk', smin=-0.01, smax=1000, nsteps=1000, color='cornflowerblue', spow=2, label='Disk')
an.plot_source_count_band('dsk', smin=-0.01, smax=1000, nsteps=1000, qs=[0.16, 0.5, 0.84], color='cornflowerblue', alpha=0.3, spow=2)
an.plot_source_count_median('gce', smin=-0.01, smax=1000, nsteps=1000, color='forestgreen', spow=2) an.plot_source_count_band('gce', smin=-0.01, smax=1000, nsteps=1000, qs=[0.16, 0.5, 0.84], color='forestgreen', alpha=0.3, spow=2)
```

along with the following `matplotlib` plotting options:

```python
plt.yscale('log')
```
Figure 1. Triangle plot obtained by analyzing the results of an NPTF in the Galactic Center, showing the one- and two-dimensional posteriors of the 11 parameters floated in the fit corresponding to three Poissonian and two non-Poissonian templates. For this analysis, 3FGL point sources have been masked at 95% containment. See the text for details.
also show a histogram of the detected 3FGL sources within the relevant energy range and region, with vertical error bars indicating the 68% confidence interval from Poisson counting uncertainties only. Since we have explicitly masked all 3FGL sources, we see that the disk- and NFW-correlated PS templates contribute at fluxes near and below the 3FGL PS detection threshold, which is $5 \times 10^{-10}$ counts cm$^{-2}$ s$^{-1}$ in this case.

### 4.5.3. Plot Intensity Fractions

The intensity fractions for the smooth and PS NFW-correlated models may be plotted with

```python
an.plot_intensity_fraction_non_poiss('gce', bins=800, color='cornflowerblue', label='GCE PS') an.plot_intensity_fraction_poiss('gce', bins=800, color='lightsalmon', label='GCE DM') plt.xlabel('Flux fraction (%)') plt.legend(fancybox=True) plt.xlim(0,6)
```

This is shown in Figure 3. We immediately see a preference for NFW-correlated point sources over the smooth NFW component.

### 4.5.4. Further Analyses

The example above may easily be pushed further in many directions, many of which are outline in (Lee et al. 2016). For example, a natural method for performing model comparison in the Bayesian framework is to compute the Bayes factor between two models. Here, for example, we may compute the Bayes factor between the model with and without NFW-correlated PSs. This involves repeating the scan described above but only adding in disk-correlated PSs. Then, by comparing the global Bayesian evidence between the two scans (see Section 3 for the syntax on how to extract the Bayesian evidence), we find a Bayes factor $\sim 10^4$ in preference for the model with spherical PSs.

Another straightforward generalization of the example described above is simply to leave out the PS mask, so that the NFW- and disk-correlated PS templates must account for both the resolved and unresolved PSs. The likelihood evaluations take longer, in this case, since there are pixels with higher photon counts compared to the 3FGL-masked scan. The result for the source-count distribution from this analysis is shown in Figure 4. In this case, the disk-correlated PS template accounts for the majority of the resolved PS emission.
5. Conclusion

We presented an open-source code package for performing non-Poissonian template fits. We strongly recommend referring to the online documentation (see footnote 7)—which will be kept up-to-date—in addition to this paper accompanying the initial release. There are many way in which NPTFit can be improved in the future. For one, the NPTFit package only handles a single energy bin at a time. In a later version of the code we plan to incorporate the ability to scan over multiple energy bins simultaneously. Additionally, there are a few areas—such as the evaluation of the incomplete gamma functions—where the Cython code may still be sped up. Such improvements to the computational cost are relevant for analyses of large data sets with many model parameters. Of course, we welcome additional suggestions for how we may improve the code and better adapt it to applications beyond the gamma-ray applications it has been used for so far.

Foremost, we thank Samuel Lee for contributing significantly not just to the original conception of the NPTF but also to an early version of the NPTFit package. We also thank Lina Necib for contributing to NPTFit. Additionally, we thank our collaborators Tim Linden, Mariangela Lisanti, Tracy Slatyer, and Wei Xue, who worked with us on projects utilizing an early version of NPTFit. We thank Douglas Finkbeiner, Dan Hooper, Christoph Weniger, and Hannes Zechlin for discussions related to the NPTF and NPTFit. NLR is supported in part by the American Australian Association’s ConocoPhillips Fellowship. B.R.S. is supported by a Pappalardo Fellowship in Physics at MIT. The work of B.R.S. was performed in part at the Aspen Center for Physics, which is supported by National Science Foundation grant PHY-1066293. This work is supported by the U.S. Department of Energy (DOE) under cooperative research agreement DE-SC-0012567 and DE-SC-0013999.

Appendix A

Mathematical Foundations of NPTFit

In this section, we present the mathematical foundation of the NPTF and the evaluation of the non-Poissonian likelihood in more detail that what was shown in Section 2. Note that many of the details presented in this section have appeared in the earlier works of (Malysh & Hogg 2011; Lee et al. 2015, 2016), however we have reproduced these here in order to have a single clear picture of the method.

The remainder of this section is divided as follows. First, we outline how to determine the generating functions for the Poisson and non-Poissonian case. We then describe how we account for finite PSF corrections.

A.1. The (non-)Poissonian Generating Function

There are two reasons why the evaluation of the Poissonian likelihood for traditional template fitting can be evaluated rapidly. The first is that the functional form of the Poissonian likelihood is simple. Second, and more importantly, is the fact that if we have two discrete random variables X and Y that follow Poisson distributions with means $\mu_1$ and $\mu_2$, then the random variable $Z = X + Y$ again follows a Poisson distribution with mean $\mu_1 + \mu_2$. This generalizes to combining an arbitrary number of random Poisson distributed variables and is why we were able to write $\mu_{p,t}(\theta) = A_t(\theta)T_{p,t}^{(S)}$ in Section 2. This fact is not true when combining arbitrary random variables, and in particular if we add in a template following non-Poissonian statistics.

An elegant solution to this problem was introduced in (Malysh & Hogg 2011), using the method of generating functions. As we are always dealing with pixelized maps containing discrete counts (of photons or otherwise), for any model of interest there will always be a discrete probability distribution $p_k$, the probability of observing $k = 0, 1, 2, \ldots$ counts. In terms of these, we then define the probability generating function as in (6). The property of probability generating functions that make them so useful in the present context is as follows. Consider two random processes X and Y, with generating functions $P_X(t)$ and $P_Y(t)$, that follow arbitrary and potentially different statistical distributions. Then the generating function of $Z = X + Y$ is simply given by the product $P_X(t) \cdot P_Y(t)$. In this subsection, we will derive the appropriate form of $P(t)$ for Poissonian and non-Poissonian statistics.

To begin with, consider the purely Poissonian case. Here and throughout this section, we consider only the likelihood in a single pixel; the likelihood over a full map is obtained from the product of the pixel-based likelihoods. Then for a Poisson distribution with an expected number of counts $\mu_p$ in a pixel $p$:

$$p_k = \frac{\mu_p^k e^{-\mu_p}}{k!}. \quad (12)$$

Note that the variation of the $\mu_p$ across the full map will be a function of the model parameters, such that $\mu_p = \mu_p(\theta)$. To simplify the notation in this section however, we leave the $\theta$ dependence implicit. Given the $p_k$ values, we then have

$$P_p(t) = \sum_{k=0}^{\infty} \frac{\mu_p^k e^{-\mu_p}}{k!} t^k$$

$$= e^{-\mu_p} \sum_{k=0}^{\infty} \frac{(\mu_p t)^k}{k!}$$

$$= \exp[\mu_p (t - 1)]. \quad (13)$$

From this form, it is clear that if we have two Poisson distributions with means $\mu_p^{(1)}$ and $\mu_p^{(2)}$, the product of their generating functions will again describe a Poisson distribution, but with mean $\mu_p^{(1)} + \mu_p^{(2)}$.

Next we work toward the generating function in the non-Poissonian case. At the outset, we let $x_{p,m}$ denote the average number of sources in a pixel $p$ that emit exactly $m$ counts. In terms of this, the probability of finding $n_m$ $m$-count sources in this pixel is just a draw from a Poisson distribution with mean $\lambda_{p,m}$, i.e.,

$$P_{n_m} = \frac{x_{p,m}^{n_m} e^{-x_{p,m}}}{n_m!}. \quad (14)$$

Given this, the probability to find $k$ counts from a population of $m$-count sources is

$$p_k^{(m)} = \begin{cases} P_{n_m}, & \text{if } k = m \cdot n_m \text{ for some } n_m, \\ 0, & \text{otherwise} \end{cases}. \quad (15)$$
We can then use this to derive the non-Poissonian m-count generating function as follows:

\[ P_{\text{NP}}^{(m)}(t) = \sum_k P_k t^k \]

\[ = \sum_{n_m} \frac{x_{p,m}^n e^{-x_{p,m}}}{n_m!} \]

\[ = \exp [x_{p,m}(t^m - 1)]. \quad (16) \]

However, this is just the generating function for m-count sources, to get the full non-Poissonian generating function we need to multiply this over all values of m. Doing so we arrive at

\[ P_{\text{NP}}(t) = \prod_{m=1}^{\infty} \exp [x_{p,m}(t^m - 1)] \]

\[ = \exp \left[ \sum_{m=1}^{\infty} x_{p,m}(t^m - 1) \right], \quad (17) \]

justifying the form given in Section 2. Again recall for the full likelihood we can just multiply the pixel-based likelihoods and that \( x_{p,m} = x_{p,m}(\theta) \).

So far, we have said nothing of how to determine \( x_{p,m} \), the average number of m-count source in pixel p. This value depends on the source-count distribution \( dN_p/dS \), which specifies the distribution of sources as a function of their expected number of counts, S. Of course the physical object is \( dN/dF \), where F is the flux. This distinction was discussed in Section 2 and can be implemented in NPTFit to arbitrary precision. Nevertheless, \( dN_p/dS \) does not fully determine \( x_{p,m} \)—we need to account for the fact that a source that is expected to give S photons could Poisson fluctuate to give m. As such any source can in principle contribute to \( x_{p,m} \), and so integrating over the full distribution we arrive at

\[ x_{p,m} = \int_0^{\infty} \frac{dN_p}{dS}(S) S^m e^{-S} m! \]. \quad (18) \]

An important part of implementing the NPTF in a rapid manner, which is a central feature of NPTFit, is the analytic evaluation of the integral in this equation. To do this, we need to have a specific form of the source-count distribution. For this purpose, we allow the source-count distribution to be a multiply broken power law and evaluate the integral for any number of breaks. The details of this calculation are presented in Appendix C.

Putting the evaluation of the integral aside for the moment then, we arrived at the full non-Poissonian generating function:

\[ P_{\text{NP}}(t) = \exp \left[ \sum_{m=1}^{\infty} x_{p,m}(t^m - 1) \right], \] \[ x_{p,m} = \int_0^{\infty} \frac{dN_p}{dS}(S) S^m e^{-S} m! \]. \quad (19) \]

Contrasting this with Equation (13), we see that while the Poissonian likelihood is specified by a single number \( \mu_p \), the non-Poissonian likelihood is instead specified by a distribution \( dN_p/dS \).

In the case of multiple PS templates, we should multiply the independent probability generating functions. However, this is equivalent to summing the \( x_{p,m} \) parameters. This is how multiple PS templates are incorporated into the NPTFit code:

\[ x_{p,m} \rightarrow \sum_{\ell=1}^{N_{\text{PS}}} x_{p,m}^{\ell} \]

\[ \text{where the sum over } \ell \text{ is over the contributions from individual PS templates.} \]

A.2. Correcting for a Finite PSF

The next factor to account for is the fact that in any realistic data set there will be a non-zero PSF. Here, we closely follow the discussion in (Malyshew & Hogg 2011). The PSF arises due to the inability of an instrument to perfectly reconstruct the original direction of the photon, neutrino, or quantity making up the counts. In practice, a finite PSF means that a source in one pixel can contribute counts to nearby pixels as well. To implement this correction, we modify the calculation of \( x_{p,m} \) given in Equation (19), which accounts for the distribution of sources as a function of S and the fact that each one could Poisson fluctuate to give m counts. The finite PSF means that in addition to this, we also need to draw from the distribution \( \rho(f) \), that determines the probability that a given source contributes a fraction of its flux \( f \) in a given pixel. Once we know \( \rho(f) \), this modifies our calculation of \( x_{p,m} \) in Equation (19)—now a source that is expected to contribute S counts, will instead contribute \( \beta S \), where \( f \) is drawn from \( \rho(f) \). As such we arrive at the result in (10).

In NPTFit, we determine \( \rho(f) \) using MC. To do this, we place a number of PSs appropriately smeared by the PSF at random positions on a pixelized sphere. Then integrating over all pixels we can determine the fraction of the flux in each pixel \( f_p \), \( p = 1, \ldots, N_{\text{pix}} \), defined such that \( f_1 + f_2 + \ldots = 1 \). Note in practice one can truncate this sum at some minimal value of \( f \) without impacting the argument below. From the set \( \{f_p\} \), we then denote by \( \Delta n(f) \) the number of fractions for \( n \) point sources that fall within some range \( \Delta f \). From these quantities, we may determine \( \rho(f) \) as

\[ \rho(f) = \lim_{\Delta f \to 0} \frac{\Delta n(f)}{n \Delta f}, \quad (21) \]

which is normalized such that \( \int df \rho(f) = 1 \). From this definition, we see that the case of a vanishing PSF is just \( \rho(f) = \delta(f - 1) \)—i.e., the flux is always completely in the pixel with the PS.

Appendix B

NPTFit: Algorithms

The generating-function formalism for calculating the probabilities \( p_{\text{NP}}^{(m)}(\theta) \) is described at the end of Section 2 and in more detail in Appendix A. In particular—given the generating function \( P(t) \)—we are instructed to calculate the probabilities by taking \( n_p \) derivatives as in (7). However, taking derivatives is numerically costly, so instead we developed recursive algorithms for computing these probabilities. In the same spirit, we analytically evaluate the \( x_{p,m} \) parameters defined in (10) for the multiply broken source-count distribution to facilitate a fast evaluation of the NPTF likelihood function. In this appendix, we overview these methods that are essential to making NPTFit a practical software package.

In general, we may write the full single pixel generating function for a model containing an arbitrary number of
Poissonian and non-Poissonian templates as

\[ P(t) = e^{f(t)}, \]

where we have defined

\[ f(t) \equiv \mu_p(t - 1) + \sum_{m=1}^{\infty} x_{p,m}(t^m - 1). \]

Above, \( x_{p,m} \) represents the average number of \( m \)-count source in pixel \( p \). The remaining task is to efficiently calculate the probabilities \( p_k \), which are formally defined in terms of derivatives through (7). Nevertheless, derivatives are slow to implement numerically, so we instead use a recursion relation to determine \( p_k \) in terms of \( p_{<k} \).

To begin with, note that

\[ p_0 = e^{f(0)}, \]
\[ p_1 = f^{(1)} e^{f(0)}. \]

For the rest of this discussion, we suppress the pixel index \( p \), though one should keep in mind that this process must be performed independently in every pixel. From (24), we can immediately write down

\[ p_k = \sum_{n=0}^{k-1} \frac{1}{k(k-n-1)!} f^{(k-n)} x_{p,n} n, \]

which as mentioned requires the knowledge of all \( p_{<k} \).

To derive (26), we first define

\[ F^{(k)}(t) \equiv \frac{d^k}{dt^k} e^{f(t)}. \]

Then, for example,

\[ F^{(1)}(t) = f^{(1)}(t) e^{f(t)}. \]

From here, to determine \( F^{(k)}(t) \), we simply need \( k - 1 \) more derivatives. Using the generalized Leibniz rule, we have

\[ F^{(k)}(t) = \sum_{n=0}^{k-1} \binom{k-1}{n} \frac{d^{k-n}}{dt^{k-n}} f^{(1)}(t) \frac{d^n}{dt^n} e^{f(t)}. \]

In the following, we will suppress the overall factor of \( T_p^{\text{PS}}/E_p \), since it does not play an important role in this discussion and may always be restored by simply rescaling \( A \). In the same spirit, we also suppress the pixel index \( p \) in \( x_{p,m} \).
With this in mind, we may explicitly evaluate the expression for \( x_{p,m} \) using (18)

\[
x_m = \frac{A}{m!} \left[ s_b^m \int_0^\infty ds \int_0^\infty ds' \, s^{m-n_1} s'^{m-n_2} \right] = \frac{A}{m!} \left[ s_b^m \Gamma(1 - n_1 + m, s_b) + \int_0^s \, s'^{m-n_2} \Gamma(1 - n_2 + m, s_b) \right]
\]

Now using our general result (31) above, we have

\[\sum_{m=1}^\infty x_m = A \left[ s_b^m \int_0^\infty ds \int_0^\infty ds' \, s^{m-n_1} s'^{m-n_2} \right] - x_{0,0} = A S_b \left[ \frac{1}{1 - n_1} + \frac{1}{1 - n_2} \right] - x_0. \]

This is useful because we already know \( x_0 \) from the general form of \( x_m \) above.

### C.2. 2 Breaks

For two breaks, the source-count distribution is given in terms of counts by

\[
dN_p \over dS = A \frac{T_p(\Psi)}{\bar{E}_p} \begin{cases} \begin{align*} S &= \frac{S_{b,1}}{S_{b,2}}, & S \geq S_{b,1} \\ \frac{S}{S_{b,1}}^{-n_1} \left( \frac{S}{S_{b,1}} \right)^{-n_2} \left( \frac{S}{S_{b,2}} \right)^{-n_3}, & S > S_{b,2}. \end{align*} \end{cases}
\]

Again suppressing the pixel-dependent prefactors, an explicit evaluation gives

\[
x_m = \frac{A S_{b,1}^m}{m!} \left[ \Gamma(1 - n_1 + m, s_{b,1}) + \frac{s_{b,1}}{s_{b,2}} \Gamma(1 - n_2 + m, s_{b,1}) \right] + \left[ \frac{s_{b,1}}{s_{b,2}} \Gamma(1 - n_3 + m, s_{b,2}) \right]
\]

\[\sum_{m=1}^\infty x_m = A S_{b,1} \left[ \frac{1}{1 - n_1} + \frac{1}{1 - n_2} \right] \times \left( 1 - \left( \frac{s_{b,2}}{s_{b,1}} \right)^{-n_3} \right) \left( \frac{s_{b,2}}{s_{b,1}} \right)^{-n_1} \right] - x_0. \]

### C.3. \( k \) Breaks

The source-count distribution in the general \( k \)-break case is given in (4) in terms of flux. In terms of counts and again suppressing pixel-dependent prefactors the result for \( x_m \) and \( \sum_{m=1}^\infty x_m \) is a simple generalization from the expressions for the one- and two-break cases:

\[
x_m = \frac{A S_{b,1}^m}{m!} \left[ \Gamma(1 - n_1 + m, s_{b,1}) + \frac{1}{\prod_{i=1}^{k} \Gamma(1 - n_{i+1} + m, s_{b,i})} \right] + \prod_{i=1}^{k} \left( \frac{s_{b,i}}{s_{b,i-1}} \right)^{-n_i} \Gamma(1 - n_{k+1} + m, s_{b,k+1}) \right]
\]

\[\sum_{m=1}^\infty x_m = A S_{b,1} \left[ \frac{1}{1 - n_1} + \frac{1}{1 - n_2} \right] \times \left( 1 - \left( \frac{s_{b,2}}{s_{b,1}} \right)^{-n_3} \right) \left( \frac{s_{b,2}}{s_{b,1}} \right)^{-n_1} \right] - x_0. \]

### References

Aartsen, M. G., Abbasi, R., Abdou, Y., et al. (IceCube) 2013a, PhRvL, 111, 021103

Aartsen, M. G., Abbasi, R., Abdou, Y., et al. (IceCube) 2013b, Sci, 342, 1242856

Aartsen, M. G., Abraham, K., Ackermann, M., et al. (IceCube) 2015a, ApJ, 809, 98

Aartsen, M. G., Abraham, K., Ackermann, M., et al. (IceCube) 2015b, PhRvL, 115, 081102

Abazajian, K. N. 2011, JCAP, 1103, 010

Abazajian, K. N., Canac, N., Horiku, S., & Kaplinghat, M. 2014, Phys. Rev. D, 90, 023526

Abazajian, K. N., Canac, N., Horiku, S., Kaplinghat, M., & Kwa, A. 2015, JCAP, 1507, 013

Abazajian, K. N., & Kaplinghat, M. 2012, PhRvD, 86, 083511

Acero, F., Ackermann, M., Ajello, M., et al. (Fermi-LAT) 2015, arXiv:1501.02003

Ackermann, M., Ajello, M., Albert, A., et al. (Fermi-LAT) 2016, PhRvL, 116, 151105

Ajello, M., Albert, A., Atwood, W. B., et al. (Fermi-LAT) 2016, ApJ, 819, 44

Bartels, R., Krishnamurthy, S., & Weniger, C. 2016, PhRvL, 116, 051102

Bechtol, K., Ahlers, M., Di Mauro, M., Ajello, M., & VandenBroucke, J. 2015, arXiv:1511.00688

Behnke, N., Bradshaw, R., Cito, C., et al. 2011, CSE, 13, 31

Boyarsky, A., Malyshev, D., & Ruchayskiy, O. 2011, PhLB, 705, 165

Brandt, T. D., & Kocsis, B. 2015, ApJ, 812, 15

Buchner, J., Georgakakis, A., Nandra, K., et al. 2014, A&A, 564, A125

Calore, F., Cholis, I., & Weniger, C. 2015, JCAP, 1503, 038

Calore, F., Di Mauro, M., Donato, F., & Donato, F. 2014, ApJ, 796, 14

Cholis, I., Hooper, D., & Linden, T. 2015, JCAP, 1506, 043

Clark, H. A., Scott, P., Trotta, R., & Lewis, G. F. 2016, arXiv:1612.01539

Daylan, T., Finkbeiner, D. P., Hooper, D., et al. 2016a, PhRvD, 12

Daylan, T., Portillo, S. K. N., & Finkbeiner, D. P. 2016b, arXiv:1607.04637

Feroz, F., & Hobson, M. P. 2008, MNRAS, 384, 449

Feroz, F., Hobson, M. P., & Bridges, M. 2009, MNRAS, 398, 1601

Feroz, F., Hobson, M. P., Cameron, E., & Pettitt, A. N. 2013, arXiv:0809.3437

Feyereisen, M. R., Ando, S., & Lee, S. K. 2015, JCAP, 1509, 027

Feyereisen, M. R., Tamborra, I., & Ando, S. 2016, arXiv:1610.01607

Foreman-Mackey, D., Voit, W., Price-Whelan, A., et al. 2016, corner.py, v2.0.0, doi:10.5281/zenodo.35155

Galassi, M., Davies, J., Theiler, J., et al. 2015, Library available online at http://www.gnu.org/software/gsl

Gendreau, K. C., Barcons, X., & Fabian, A. C. 1998, MNRAS, 297, 41

Georgantopoulos, I., Stewart, G. C., Shanks, T., Griffiths, R. E., & Boyle, B. J. 1993, MNRAS, 262, 619

Goodenough, L., & Hooper, D. 2009, arXiv:0910.2998
Gordon, C., & Macias, O. 2013, PhRvD, 88, 083521
Gorski, K. M., Hivon, E., Banday, A. J., et al. 2005, ApJ, 622, 759
Hasinger, G., Burg, R., Giacconi, R., et al. 1993, A&A, 275, 1
Hooper, D., Cholis, I., Linden, T., Siegal-Gaskins, I., & Slatyer, T. R. 2013, PhRvD, 88, 083009
Hooper, D., & Goodenough, L. 2011, PhLB, 697, 412
Hooper, D., & Linden, T. 2011, PhRvD, 84, 123005
Hooper, D., & Slatyer, T. R. 2013, PDU, 2, 118
Hunter, J. D. 2007, CSE, 9, 90
Johansson, F., et al. 2013, mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18), http://mpmath.org/
Lee, S. K., Lisanti, M., & Safdi, B. R. 2015, JCAP, 1505, 056
Lee, S. K., Lisanti, M., Safdi, B. R., Slatyer, T. R., & Xue, W. 2016, PhRvL, 116, 051103
Linden, T., Rodd, N. L., Safdi, B. R., & Slatyer, T. R. 2016, arXiv:1604.01026
Lisanti, M., Mishra-Sharma, S., Ncib, L., & Safdi, B. R. 2016, arXiv:1606.04101
Macias, O., Gordon, C., Crocker, R. M., et al. 2016, arXiv:1611.06644
Malyshev, D., & Hogg, D. W. 2011, ApJ, 738, 181
Miyaji, T., & Griffiths, R. E. 2002, ApJL, 564, L5
Murase, K., & Waxman, E. 2016, MNRAS, arXiv:1607.01601
O’Leary, R. M., Kisler, M. D., Kerr, M., & Dexter, J. 2015, arXiv:1504.02477
Olphant, T. E. 2006, A Guide to NumPy, Vol. 1 (USA: Trelgol Publishing)
Pérez, F., & Granger, B. E. 2007, CSE, 9, 21
Perri, M., & Giommi, P. 2000, A&A, 362, L57
Petrović, J., Serpico, P. D., & Zaharijas, G. 2015, JCAP, 1502, 023
Sklenging, J. 2006, BoyAn, 1, 833
Su, M., Slatyer, T. R., & Finkbeiner, D. P. 2010, ApJ, 724, 1044
Yuan, Q., & Ioka, K. 2015, ApJ, 802, 124
Zechlin, H.-S., Cuoco, A., Donato, F., Fornengo, N., & Regis, M. 2016b, ApJL, 826, L31
Zechlin, H.-S., Cuoco, A., Donato, F., Fornengo, N., & Vittino, A. 2016a, ApJS, 225, 18