ITERATIVE UNFOLDING REFERENCE

Towards understanding iterative Bayesian unfolding

February 1, 2018

Revision 1.0

Iterative Unfolding Reference

Zigfried Hampel-Arias

February 1, 2018

Abstract

This reference outlines the D'Agostini procedure used for iterative Bayesian unfolding. We first motivate and outline the method in a manner suitable for users new to the technique. Next is shown the full propagation of errors due to the unfolding process, including the derivation of the final form of the covariance matrix. Finally, sample code in the Python language and example inputs are provided as a demonstration of proper implementation.

Contents

1 Introduction

The general class of unfolding methods is amongst the physicist's toolbox as a powerful means to connect an experiment's observable variables with true physical quantities. Typically a matrix can be built to encompass the effects of the measurement process on a simulated 'true' distribution and the manifestation of said distribution as an experimenter's desired observable. With this response matrix, a distribution of the observable in an experiment can be **unfolded**, providing an estimate of the true parent distribution.

A variety of unfolding methods exist, each with its respective strengths and weaknesses. For example, the simplest method is the matrix inversion unfolding, which for a well populated, highly linear response matrix can be both efficient and precise. However, even with relatively small off-diagonal elements, this method can be unfavorable, as the matrix may be singular or may introduce wildly fluctuating results due to limited statistics. There exist methods to quell such issues, though these require the tuning of various parameters which typically have no physical connection to the experiment at hand.

Here we discuss D'Agostini's Bayesian unfolding technique presented in [1], a manifestly **inferential** method. Starting from Bayes' theorem, an iterative unfolding procedure is developed, which then can be implemented without too much difficulty for the typical experimenter. This document has been adapted from Chapter 7 and Appendix B of [2].

2 D'Agostini Unfolding

2.1 Method

As discussed in the Section 1, the conceptually simplest way to connect true (causes, C_μ) and observable (effects, E_j) variables is via a matrix, R, and it's inverse M[1]:

$$\begin{aligned} n(E) &= R\,\phi(C), \\ \phi(C) &= M\,n(E). \end{aligned} \tag{1}$$

Due to the aforementioned potential difficulties in matrix inversion, we can take into consideration Bayes' theorem,

$$P(C_\mu|E_j) = \frac{P(E_j|C_\mu)\,P(C_\mu)}{\sum_\nu^{n_C} P(E_j|C_\nu)\,P(C_\nu)}\,, \tag{2}$$

where n_C is the number of possible causes. Equation 2 dictates that having observed the effect E_j, the probability that it's origin is due to the cause C_μ is proportional to product of the probability of the cause and the probability of the cause to produce that effect. Hence, the elements $P(E_i|C_\mu)$ represent the probability that a given C_μ results in the effect E_i, and is the response matrix typically generated via modeling or simulation. Continuing with $P(C_\mu|E_j)$, we can then connect the measured observed effects to their causes by

$$\phi(C_\mu) = \sum_i^{n_E} P(C_\mu|E_i)\,n(E_i)\,. \tag{3}$$

Stepping back to eq. 2 for a moment, one identifies $P(C_\mu)$ as the prior cause distribution, representing our current knowledge of the causes. The prior is a normalized distribution such that $\sum_\mu^{n_C} P(C_\mu) = 1$. This normalization requirement is not imposed on the response matrix efficiency ϵ_μ: $0 \leq \epsilon_\mu = \sum_j^{n_E} P(E_j|C_\mu) \leq 1$, ie, a cause does not need to produce any effect. Taking this (in)-efficiency into account, we rewrite eq. 3 as

$$\phi(C_\mu) = \frac{1}{\epsilon_\mu}\sum_i^{n_E} P(C_\mu|E_i)\,n(E_i)\,. \tag{4}$$

Identifying here the explicit form of M, the full matrix (Bayesian) inversion equation is then

$$\phi(C_\mu) = \sum_j^{n_E} M_{\mu j}\,n(E_j)\,, \tag{5}$$

where

$$M_{\mu j} = \frac{P(E_i|C_\mu)\,P(C_\mu)}{[\sum_k^{n_E} P(E_k|C_\mu)][\sum_\nu^{n_C} P(E_i|C_\nu)\,P(C_\nu)]}\,. \tag{6}$$

The response matrix $P(E_i|C_\mu)$ is generated via simulation, and the $n(E_i)$ provided through measurement, apparently bestowing the freedom to choose the form of $P(C_\mu)$. Again, $P(C_\mu)$

[1] Except for C and E, all variables and subscripts related to causes are Greek letters, while Latin letters are used for effects. The only superscript is the iteration number, i.

represents the total of our prior knowledge of the parent distribution. Typically an experimenter refrains from introducing bias in the prior so an appropriate choice is the Jeffreys Prior [6]:

$$P_{Jeffrey}(C_\mu) = \frac{1}{log(C_{max}/C_{min})\, C_\mu}\,,$$

keeping in mind that the this prior dictates that all cause bins are of equal probability, not that all parent distributions are of equal probability.

We now possess all the necessary machinery to perform an unfolding. Having started with the conservative Jeffreys Prior, the unfolded result is a Bayesian best estimate of the true distribution. There is nothing stopping us from using this result as the best knowledge estimate of $P(C_\mu)$ in eq. 6 for a subsequent unfolding. We can take this any number of steps further, making the process an iterative unfolding. Thus, after calculating $\phi(C_\mu)$ via eq. 5, we recalculate $M_{\mu j}$ per eq. 6, returning again to eq. 5 for an updated $\phi(C_\mu)'$. Since $P(C_\mu) = \frac{\phi_\mu}{\sum_\nu \phi_\nu} = \frac{\phi_\mu}{N_{True}}$, where N_{True} is the estimated true number of cause events, we can make the change $P(C_\mu) \to \phi_\mu$ in eq. 6. Adding the iteration superscript and shortening the notation[1], this equates to

$$\begin{aligned} M_{\mu j} &= \frac{P_{\mu j}\, \phi^i_\mu}{\epsilon_\mu \sum_\rho P_{\rho j}\, \phi^i_\rho} \\ \phi^{i+1}_\mu &= \sum_j M_{\mu j}\, n_j. \end{aligned}$$

The unfolding proceeds until a desired stopping criterion is satisfied, say by comparing subsequent iterations with a test statistic such as a χ^2. The algorithm below outlines the basics to the iterative unfolding scheme:

Algorithm 1 Unfolding Algorithm

```
$\phi^0 \leftarrow$ Prior
testStatistic$\leftarrow$ Pass
while ( testStatistic = Pass ) do
    $M \leftarrow M(P(E|C), \phi^i)$
    $\phi^{i+1} \leftarrow M \times n$
    testStatistic$\leftarrow$ TS$(\phi^i, \phi^{i+1})$
end while
```

2.2 Regularization

After each iteration, the resulting posterior distribution, $P(C_\mu)$, is our new best guess of the (normalized) parent distribution. Using this best estimate as the prior for the next iteration, one can induce large fluctuations in neighboring C_μ bins. It is here the equivalence of matrix inversion techniques and Bayesian unfolding is seen. After many iterations, wild fluctuations can appear, indicating the granularity in the MC derived $P_{\mu j}$. Furthermore, in using the posterior as the subsequent prior, one is 'telling' the unfolding that physical distributions of that nature are allowable priors. Instead, as pointed out in [1] (section 6.3), for an experimenter interested in a particular model's parameters, fitting all but the last posterior is equivalent to performing a maximum likelihood fit to the data.

As physics measurements are expected to be smooth (a safe assumption for the cosmic-ray energy spectrum for example), one can regularize the ϕ^i_μ. In principle one can choose any smoothing function. For the cosmic-ray energy spectrum, ϕ^i_μ can be simply fit to a power law, using the fitted function as the input prior for the next iteration. While this could be seen as a loss of information, it is important to remember that this is a Bayesian method, so **any** improved prior distribution will enhance our estimation method, along with the **prior** expectation that our physics is smooth.

The other possibility is to avoid regularization altogether, and instead ensure that $P_{\mu j}$ is smooth enough. The granularity of the cause and effect bins will dictate the degree of smoothness required to ensure non-fluctuating ϕ^i solutions. The more widely used techniques for smoothing $P_{\mu j}$ include kernel density estimation and penalized spline fitting routines.

2.3 Unfolding Uncertainties

To begin the excursion into the calculation of uncertainties, we first shorten the notation in accordance with footnote [1]:

$$P(E_i|C_\mu) = P_{\mu i} \qquad\qquad \phi(C_\mu) = \phi_\mu \qquad\qquad n(E_j) = n_j.$$

As outlined in [1] (section 4), the covariance matrix $V = V(\phi, \phi')$ from statistical contributions has two components: V^{Data} from the counted measured effects distribution, and V^{MC} due to the limited MC statistics in $P_{\mu j}$. This can be seen from considering the uncertainties from n_j and $M_{\mu j}$ in eq. 5. Since $\phi = M \times n = M(P(E|C)) \times n$, we can identify respectively the aforementioned error contributions as

$$\begin{aligned} V^{Total} &= V^{Data} + V^{MC} \\ &= \frac{\partial \phi}{\partial n}\, Cov(n, n')\, \frac{\partial \phi'}{\partial n} \\ &+ \frac{\partial \phi}{\partial P}\, Cov(P, P')\, \frac{\partial \phi'}{\partial P}. \end{aligned}$$

2.3.1 V^{Data}

D'Agostini argues that since the data sample n_j is a realization of a multinomial distribution, then

$$V^{Data} = M\, Cov(n, n')\, M \tag{7}$$

where the $Cov(n, n')$ is the covariance matrix of the measurements with respect to the estimated true number of events $\sum_\mu \phi_\mu = N_{true}$:

$$Cov(n_k, n_j) = \begin{cases} n_j(1 - \frac{n_j}{N_{true}}) & \text{if } k = j \\ -\frac{n_j n_k}{N_{true}} & \text{if } k \neq j \end{cases}. \tag{8}$$

However, Adye ([3] section 5) demonstrates that this error estimation is only valid for the first iteration, as subsequent ϕ^i are **not independent** of n_j. Indeed, we should re-write eq. 7

appropriately as

$$V^{Data} = \frac{\partial \phi^{i+1}}{\partial n} \times Cov(n, n') \times \frac{\partial \phi^{i+1\prime}}{\partial n}, \tag{9}$$

with

$$\frac{\partial \phi^{i+1}_{\mu}}{\partial n_j} = M_{\mu j} + \frac{\phi^{i+1}_{\mu}}{\phi^{i}_{\mu}} \frac{\partial \phi^{i}_{\mu}}{\partial n_j} - \sum_{\sigma,k} \epsilon_{\sigma} \frac{n_k}{\phi^{i}_{\sigma}} M_{\mu k} M_{\sigma k} \frac{\partial \phi^{i}_{\sigma}}{\partial n_j}$$

where again the superscripts i and $i+1$ refer to the iteration number. The full derivation of $\frac{\partial \phi^{i+1}}{\partial n}$ (eq. 20) is found in section 2.4.2 below. In some cases it is safe to use the Poisson form of $Cov(n, n')$:

$$Cov(n_k, n_j) = n_k \, \delta_{kj} \, . \tag{10}$$

2.3.2 V^{MC}

The contribution from V^{MC}, while well outlined in [1] and below, is quite a monster. If one simply implements the equation verbatim into code, the expected time for calculating all elements $\sim$ (number of bins)^7. Thus, here we present the form of V^{MC}, while in section 2.4.3 we show the explicit expansion and further contraction of indices towards a more reasonable, practical calculation.

D'Agostini identifies V^{MC} via $\frac{\partial}{\partial M}$ giving

$$V^{MC} = n \times Cov(M, M') \times n'. \tag{11}$$

Further expansion reveals

$$Cov(M_{\mu k}, M_{\lambda j}) = \sum_{\{\sigma r\},\{\sigma s\}} \frac{\partial M_{\mu k}}{\partial P_{\sigma r}} \frac{\partial M_{\lambda j}}{\partial P_{\sigma s}} Cov(P_{\sigma r}, P_{\sigma s}), \tag{12}$$

$$\frac{\partial M_{\mu k}}{\partial P_{\sigma j}} = M_{\mu k} \left[\frac{\delta_{\mu\sigma} \, \delta_{jk}}{P_{\sigma j}} - \frac{\delta_{\mu\sigma}}{\epsilon_{\sigma}} - \frac{\delta_{jk} \, M_{\sigma k} \, \epsilon_{\sigma}}{P_{\sigma k}} \right], \tag{13}$$

$$Cov(P_{\sigma r}, P_{\sigma s}) = \begin{cases} \frac{1}{\tilde{n}_{\sigma}} P_{\sigma r} \, (1 - P_{\sigma r}) & \text{if } r = s \\ -\frac{1}{\tilde{n}_{\sigma}} P_{\sigma r} \, P_{\sigma s} & \text{if } r \neq s \end{cases}. \tag{14}$$

In the final expression, $\tilde{n}_{\mu}$ represents the number of simulated events which fell into the true cause bin μ. If our simulation is weighted, we identify $\tilde{n}$ with the effective number of events $\tilde{n}_{\mu} = \frac{(\sum_j w_{\mu j})^2}{\sum_j w^2_{\mu j}}$ for all j events in bin μ.

Once again, Adye ([4]) shows this is a first order estimate, only valid for the first iteration. Re-writing 11 with $\frac{\partial}{\partial P}$,

$$V^{MC} = \frac{\partial \phi^{i+1}}{\partial P} \times Cov(P, P') \times \frac{\partial \phi^{i+1\prime}}{\partial P}, \tag{15}$$

we identify $\frac{\partial \phi^{i+1}}{\partial P}$ as

$$\frac{\partial \phi^{i+1}_{\mu}}{\partial P_{\lambda k}} = \frac{\delta_{\lambda\mu}}{\epsilon_{\mu}} \left(\frac{n_k \, \phi^{i}_{\mu}}{f_k} - \phi^{i+1}_{\mu} \right) - \frac{n_k \phi^{i}_{\lambda}}{f_k} M_{\mu k}$$
$$+ \frac{\phi^{i+1}_{\mu}}{\phi^{i}_{\mu}} \frac{\partial \phi^{i}_{\mu}}{\partial P_{\lambda k}} - \sum_{\rho,j} n_j \frac{\epsilon_{\rho}}{\phi^{i}_{\rho}} M_{\rho j} \, M_{\mu j} \frac{\partial \phi^{i}_{\rho}}{\partial P_{\lambda k}}$$

whose derivation (eq. 21) is found in section 2.4.3 below. Of course, D'Agostini's form of $Cov(P, P')$ remains valid for use with the new construction of the partials. One may also use a Poisson covariance if justified appropriately:

$$Cov(P_{\rho r}, P_{\lambda s}) = \sigma_{\rho r}\sigma_{\lambda s}\delta_{\rho\lambda}\delta_{rs}, \tag{16}$$

with $\sigma_{\rho r}$ being the error estimates on $P_{\rho r}$ estimated when filling P with Monte Carlo.

2.3.3 Updated Unfolding Algorithm

The afore-outlined unfolding algorithm must be modified to include the propagation of systematic errors. At each iteration we have ϕ^{i+1}, so both $\frac{\partial\phi^{i+1}}{\partial n}$ and $\frac{\partial\phi^{i+1}}{\partial P}$ can be calculated. The results are propagated and saved until the full covariance matrix is required for error estimates on the final ϕ.

Algorithm 2 Unfolding Algorithm - Including Errors

```
$\phi^0 \leftarrow$ Prior
testStatistic$\leftarrow$ Pass
while ( testStatistic = Pass ) do
    $M \leftarrow M(P(E|C), \phi^i)$
    $\phi^{i+1} \leftarrow M \times n$
    $\frac{\partial\phi^{i+1}}{\partial n} \leftarrow$ eq. 20
    $\frac{\partial\phi^{i+1}}{\partial P} \leftarrow$ eq. 21
    testStatistic$\leftarrow$ TS$(\phi^i, \phi^{i+1})$
end while
$V^{Total} \leftarrow V^{Data}(\frac{\partial\phi^{i+1}}{\partial n}) + V^{MC}(\frac{\partial\phi^{i+1}}{\partial P})$
$\sigma_\phi^2 \approx diag(V^{Total})$
```

2.4 Expansion of Components of V

2.4.1 Some useful formulae

Recalling the unfolding formulae from before,

$$\phi_\mu^{i+1} = \sum_k M_{\mu k}\, n_k \qquad\qquad M_{\mu j} = \frac{P_{\mu j}\,\phi_\mu^i}{\epsilon_\mu\, f_j},$$

where the efficiency, ϵ, and normalization, f, of M are

$$\epsilon_\mu = \sum_j P_{\mu j} \qquad\qquad f_j = \sum_\mu P_{\mu j}\,\phi_\mu^i.$$

Of note is the presence of ϕ^i, ie, the unfolded cause distribution from the previous iteration, or the prior in the case $i = 0$.

We will be taking derivatives of these objects with respect to n_k and $P_{\lambda k}$, to wit,

$$\frac{\partial P_{\mu j}}{\partial n_k} = 0 \qquad \frac{\partial \epsilon_\mu}{\partial n_k} = 0 \qquad \frac{\partial f_j}{\partial n_k} = \sum_\mu P_{\mu j} \frac{\partial \phi^i_\mu}{\partial n_k} \tag{17}$$

$$\frac{\partial P_{\mu j}}{\partial P_{\lambda k}} = \delta_{\mu\lambda}\,\delta_{jk} \qquad \frac{\partial \epsilon_\mu}{\partial P_{\lambda k}} = \delta_{\lambda\mu} \qquad \frac{\partial f_j}{\partial P_{\lambda k}} = \delta_{jk}\,\phi^i_\lambda + \sum_\mu P_{\mu j} \frac{\partial \phi^i_\mu}{\partial P_{\lambda k}}. \tag{18}$$

The explicit forms of $\frac{\partial \phi^i_\mu}{\partial n_k}$ and $\frac{\partial \phi^i_\mu}{\partial P_{\lambda k}}$ will be shown below, but only for $i = 0$ do

$$\frac{\partial \phi^i_\mu}{\partial n_k} = 0 \quad , \quad \frac{\partial \phi^i_\mu}{\partial P_{\lambda k}} = 0, \tag{19}$$

as no unfolding has been performed. This will clearly not be the case for subsequent iterations when ϕ^i becomes dependent on n_k and $P_{\lambda k}$.

2.4.2 Expansion of V^{Data}

Making the appropriate substitutions, the index representation of eq. 9 is

$$V(\phi^{i+1}_\mu, \phi^{i+1}_\nu)^{Data} = \sum_{jk} \frac{\partial \phi^{i+1}_\mu}{\partial n_j}\, Cov(n_j, n_k)\, \frac{\partial \phi^{i+1}_\nu}{\partial n_k},$$

with

$$\begin{aligned}
\frac{\partial \phi^{i+1}_\mu}{\partial n_j} &= \frac{\partial}{\partial n_j} \sum_k M_{\mu k}\, n_k \\
&= \sum_k (M_{\mu k} \frac{\partial n_k}{\partial n_j} + n_k \frac{\partial M_{\mu k}}{\partial n_j}) \\
&= \sum_k (M_{\mu k}\, \delta_{jk} + n_k \frac{\partial M_{\mu k}}{\partial n_j}) \\
&= M_{\mu j} + \sum_k n_k \underbrace{\frac{\partial M_{\mu k}}{\partial n_j}}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial M_{\mu k}}{\partial n_j} &= \frac{\partial}{\partial n_j} \frac{P_{\mu k}\phi^i_\mu}{\epsilon_\mu f_k} \\
&= \underbrace{\frac{P_{\mu k}}{\epsilon_\mu f_k}}_{\frac{M_{\mu k}}{\phi^i_\mu}} \frac{\partial \phi^i_\mu}{\partial n_j} - \underbrace{\frac{P_{\mu k}\phi^i_\mu}{\epsilon_\mu f_k}}_{M_{\mu k}} \frac{1}{f_k} \sum_\sigma P_{\sigma k} \frac{\partial \phi^i_\sigma}{\partial n_j} \\
&= \frac{M_{\mu k}}{\phi^i_\mu} \frac{\partial \phi^i_\mu}{\partial n_j} - M_{\mu k} \sum_\sigma \epsilon_\sigma \underbrace{\frac{P_{\sigma k}}{\epsilon_\sigma f_k}}_{\frac{M_{\sigma k}}{\phi^i_\sigma}} \frac{\partial \phi^i_\sigma}{\partial n_j} \\
&= \frac{M_{\mu k}}{\phi^i_\mu} \frac{\partial \phi^i_\mu}{\partial n_j} - \sum_\sigma \frac{\epsilon_\sigma}{\phi^i_\rho} M_{\mu k} M_{\sigma k} \frac{\partial \phi^i_\sigma}{\partial n_j}
\end{aligned}$$

$$\frac{\partial\phi_\mu^{i+1}}{\partial n_j} = M_{\mu j} + \frac{1}{\phi_\mu^i}\frac{\partial\phi_\mu^i}{\partial n_j}\underbrace{\sum_k M_{\mu k} n_k}_{\phi_\mu^{i+1}} - \sum_{\sigma,k}\epsilon_\sigma \frac{n_k}{\phi_\rho^i} M_{\mu k} M_{\sigma k}\frac{\partial\phi_\sigma^i}{\partial n_j}$$

$$\frac{\partial\phi_\mu^{i+1}}{\partial n_j} = M_{\mu j} + \frac{\phi_\mu^{i+1}}{\phi_\mu^i}\frac{\partial\phi_\mu^i}{\partial n_j} - \sum_{\sigma,k}\epsilon_\sigma \frac{n_k}{\phi_\sigma^i} M_{\mu k} M_{\sigma k}\frac{\partial\phi_\sigma^i}{\partial n_j} \tag{20}$$

Recalling eq. 19, $\frac{\partial\phi_\mu^0}{\partial n_j} = 0$ for the first iteration, eliminating the last two terms of eq. 20 and recovering $\frac{\partial\phi_\mu^1}{\partial n_j} = M_{\mu j}$ as per [1]. In practice, one need only calculate $\frac{\partial\phi_\mu^{i+1}}{\partial n_j}$ for each iteration, saving the result until the full calculation of V^{Data} is required.

2.4.3 Expansion of V^{MC}

Similar to $V(\phi_\mu^{i+1}, \phi_\nu^{i+1})^{Data}$, we identify the contributions to V from the Monte Carlo:

$$V(\phi_\mu^{i+1}, \phi_\nu^{i+1})^{MC} = \sum_{\lambda j}\sum_{\rho k} \frac{\partial\phi_\mu^{i+1}}{\partial P_{\lambda j}} Cov(P_{\lambda j}, P_{\rho k}) \frac{\partial\phi_\nu^{i+1}}{\partial P_{\rho k}}.$$

Proceeding forward,

$$\frac{\partial\phi_\mu^{i+1}}{\partial P_{\lambda k}} = \frac{\partial}{\partial P_{\lambda k}}\sum_j M_{\mu j} n_j = \sum_j n_j \underbrace{\frac{\partial M_{\mu j}}{\partial P_{\lambda k}}}$$

$$\begin{aligned}
\frac{\partial M_{\mu j}}{\partial P_{\lambda k}} &= \frac{\partial}{\partial P_{\lambda k}} \frac{P_{\mu j}\phi_\mu^i}{\epsilon_\mu f_j} \\
&= \frac{\phi_\mu^i}{\epsilon_\mu f_j}\frac{\partial P_{\mu j}}{\partial P_{\lambda k}} + \underbrace{\frac{P_{\mu j}}{\epsilon_\mu f_j}}_{\frac{M_{\mu j}}{\phi_\mu^i}}\frac{\partial\phi_\mu^i}{\partial P_{\lambda k}} - \frac{1}{\epsilon_\mu f_j}\underbrace{\frac{P_{\mu j}\phi_\mu^i}{\epsilon_\mu f_j}}_{M_{\mu j}}\left(f_j\frac{\partial\epsilon_\mu}{\partial P_{\lambda k}} + \epsilon_\mu\frac{\partial f_j}{\partial P_{\lambda k}}\right) \\
&= \frac{\phi_\mu^i}{\epsilon_\mu f_j}\delta_{\lambda\mu}\delta_{jk} + \frac{M_{\mu j}}{\phi_\mu^i}\frac{\partial\phi_\mu^i}{\partial P_{\lambda k}} - \frac{1}{\epsilon_\mu f_j}M_{\mu j}\left(f_j\delta_{\lambda\mu} + \epsilon_\mu\delta_{jk}\phi_\lambda^i + \epsilon_\mu\sum_\rho P_{\rho j}\frac{\partial\phi_\rho^i}{\partial P_{\lambda k}}\right) \\
&= \frac{\phi_\mu^i}{\epsilon_\mu f_j}\delta_{\lambda\mu}\delta_{jk} + \frac{M_{\mu j}}{\phi_\mu^i}\frac{\partial\phi_\mu^i}{\partial P_{\lambda k}} - \frac{M_{\mu j}}{\epsilon_\mu}\delta_{\lambda\mu} - \frac{M_{\mu j}\phi_\lambda^i}{f_j}\delta_{jk} - \sum_\rho \epsilon_\rho M_{\mu j}\underbrace{\frac{P_{\rho j}}{\epsilon_\rho f_j}}_{\frac{M_{\rho j}}{\phi_\rho^i}}\frac{\partial\phi_\rho^i}{\partial P_{\lambda k}} \\
&= \frac{\phi_\mu^i}{\epsilon_\mu f_j}\delta_{\lambda\mu}\delta_{jk} + \frac{M_{\mu j}}{\phi_\mu^i}\frac{\partial\phi_\mu^i}{\partial P_{\lambda k}} - \frac{M_{\mu j}}{\epsilon_\mu}\delta_{\lambda\mu} - \frac{M_{\mu j}\phi_\lambda^i}{f_j}\delta_{jk} - \sum_\rho M_{\rho j}M_{\mu j}\frac{\epsilon_\rho}{\phi_\rho^i}\frac{\partial\phi_\rho^i}{\partial P_{\lambda k}},
\end{aligned}$$

and going back to $\frac{\partial\phi_\mu^{i+1}}{\partial P_{\lambda k}}$ to include the sum over j,

$$\frac{\partial \phi_\mu^{i+1}}{\partial P_{\lambda k}} =$$

$$\begin{aligned}
&\sum_j n_j \Big[\frac{\phi_\mu^i}{\epsilon_\mu f_j}\delta_{\lambda\mu}\delta_{jk} + \frac{M_{\mu j}}{\phi_\mu^i}\frac{\partial \phi_\mu^i}{\partial P_{\lambda k}} - \frac{M_{\mu j}}{\epsilon_\mu}\delta_{\lambda\mu} - \frac{M_{\mu j}\phi_\lambda^i}{f_j}\delta_{jk} - \sum_\rho M_{\rho j}M_{\mu j}\frac{\epsilon_\rho}{\phi_\rho^i}\frac{\partial \phi_\rho^i}{\partial P_{\lambda k}}\Big] \\
&= \frac{n_k\phi_\mu^i}{\epsilon_\mu f_k}\delta_{\lambda\mu} + \frac{1}{\phi_\mu^i}\frac{\partial \phi_\mu^i}{\partial P_{\lambda k}}\underbrace{\sum_j M_{\mu j}n_j}_{\phi_\mu^{i+1}} - \frac{\delta_{\lambda\mu}}{\epsilon_\mu}\underbrace{\sum_j M_{\mu j}n_j}_{\phi_\mu^{i+1}} - \frac{n_k M_{\mu k}\phi_\lambda^i}{f_k} \\
&\quad - \sum_j\sum_\rho n_j \frac{\epsilon_\rho}{\phi_\rho^i} M_{\rho j}M_{\mu j}\frac{\partial \phi_\rho^i}{\partial P_{\lambda k}},
\end{aligned}$$

with final form

$$\begin{aligned}
\frac{\partial \phi_\mu^{i+1}}{\partial P_{\lambda k}} &= \frac{\delta_{\lambda\mu}}{\epsilon_\mu}\Big(\frac{n_k\phi_\mu^i}{f_k} - \phi_\mu^{i+1}\Big) - \frac{n_k\phi_\lambda^i}{f_k}M_{\mu k} \\
&+ \frac{\phi_\mu^{i+1}}{\phi_\mu^i}\frac{\partial \phi_\mu^i}{\partial P_{\lambda k}} - \sum_{\rho,j} n_j \frac{\epsilon_\rho}{\phi_\rho^i}M_{\rho j}M_{\mu j}\frac{\partial \phi_\rho^i}{\partial P_{\lambda k}}.
\end{aligned} \tag{21}$$

Again for the first iteration $\frac{\partial \phi_\mu^0}{\partial P_{\lambda k}} = 0$, eliminating the last two terms of eq. 21, and recovering D'Agostini's version. Again, upon implementation one need only calculate $\frac{\partial \phi_\mu^{i+1}}{\partial P_{\lambda k}}$ at each iteration, saving it until V^{MC} is needed for error estimation.

3 Python Implementation

3.1 PyUnfold: Unfolding Made Simple - But Not Simpler

The PyUnfold package is available for checkout on GitHub. Its main dependencies are `matplotlib`, `pyROOT`, and `numpy`. It is rather simple to implement a spline regularization routine, simply ask the author how. To run an unfolding from the command line is simple:

```
python Unfold.py -c config.cfg -p
```

This will unfold a data set with response matrix, prior, and other input parameters all specified in the configuration file (via the `-c` flag). An output ROOT [5] file, also specified in `config.cfg`, will be generated with all pertinent details of the unfolding, including the input effects distribution, all cause distribution iterations, and the test statistic and regularization parameter values for each iteration. The `-p` flag plots the input effects distribution, a subset of unfolded distribution iterations, and the final unfolded cause distribution.

One can call the unfolding function from within other python scripts as well. This can be useful if one desires to unfold an effects distribution subject to different response functions, or perhaps different input distributions with the same response matrix, for example. To implement this, the `Unfold.py` script is first imported, then the main unfolding function can be called as:

```python
# import the unfolding project and utils
from Unfold import Unfold
from Utils import *

# grab the effects distribution
# with errors and axis properties
# as numpy arrays here

# set the RecoDist object here to pass to unfolder
RecoDist = DataDist("Reco",data=data,error=error,axis=cause,edges=cause_edges,
                     xlabel="Reco Var",ylabel="Freq.",units="Arb Units")

# do the unfolding
CauseDist = Unfold(config_name="config.cfg",return_dists=True,EffDist=RecoDist)

# grab the unfolded distribution and errors
unf_dist = CauseDist.getData()
unf_err = CauseDist.getError()
```

Of course, in this example, the script would be located in the PyUnfold project directory in order to import `Unfold` and `Utils`. For clarity, `RecoDist` is an instance of the class `DataDist` found in `Utils.py`, which gets passed to the `Unfold` function. The input variables `data`, `error`, `axis`, `edges` are all assigned to numpy arrays containing their respective namesakes. Hence, the bin centers for the `data` distribution are defined by `axis`, and the bin edges by `edges`. Finally, the unfolded distribution and respective error numpy arrays are accessed as shown in the final two lines of the above example.

If one specifically is making an **energy** dependent flux measurement, the `FluxFit.py` script can be used to account for the energy bin spacing to calculate a differential flux from the unfolded counts distribution. The command command line with all options is

```
python FluxFit.py -c config.cfg -p -s 2.6 -w
```

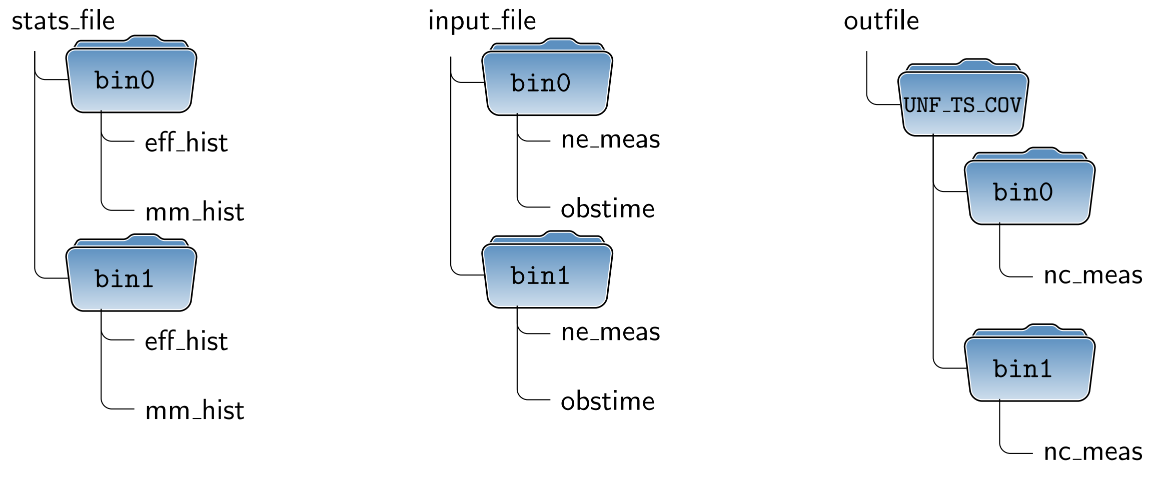
where `-p` plots the differential flux scaled by the value given to `-s`, and the resulting flux values and fit parameters (function defined in the config file) are written to the same unfolding output ROOT file with `-w`.

3.2 ROOT

A quick word on ROOT. The PyUnfold package relies on some tools provided by ROOT and pyROOT. These are **requirements** to using PyUnfold.

Another requirement is the structure of ROOT files. As shown in the next section, data structures are stored as histograms in a specific directory structure. When generating a ROOT file with the efficiency histogram and response matrix, they **must** be located inside a directory labeled `binX` where `X` is an integer identifying the bin number. Similarly, the corresponding data set to unfold must be in a directory with the same name in its ROOT file. This ensures one can use different analysis cuts defining differnet bins and pack them orderly in respective data and response files. Apart from this, one can name the ROOT objects as one wishes, so long as these labels are reflected in the configuration file.

For two analysis bins, the directories for the three ROOT files defined in the configuration file follow this structure:



3.3 Configuration File

The configuration file contains many keyword sections and options, however, only a few parameters need specification to get started. The following sections are from a complete config file which is included in the GitHub project. Each header section is presented in detail below. For beginners, the most relevant sections are `data`, `mcinput`, `output`, and `regularization`. Other sections have default values set for most purposes.

```
[data]
# File containing effects dist: N(E)
inputfile = PATH_TO_INPUT_ROOT_FILE/INPUT_FILE.root
ne_meas = Energy
obstime = TIME
ismc = False
nc_thrown = NC
```

The `data` section corresponds to the extracted data effects distribution. In practice, one will only modify the first two options: the path to the ROOT input data file (`inputfile`), and the name of the measured effects histogram to unfold (`ne_meas`). The `obstime` option identifies the ROOT histogram containing a single bin the total observation time in seconds. If one is unfolding a Monte Carlo generated data set, then the `ismc` flag must be set to `True` and the appropriate cause distribution (`nc_thrown`) can be identified.

```
[mcinput]
# File containing MC params: Eff(C), P(E|C)
stats_file = PATH_TO_RESPONSE_ROOT_FILE/RESPONSE_FILE.root
eff_hist = Eff
mm_hist = MM
```

The `mcinput` section corresponds to the response function model. The `stats_file` key identifies the path to the ROOT file containing the histograms for the 1D efficiency, `eff_hist`, and 2D mixing matrix, `mm_hist`.

```
[output]
write_output = True
outfile = PATH_TO_OUTPUT_ROOT_FILE/UNFOLD_FILE.root
nc_meas = NC
```

The `output` section details whether to write the output to file (`write_output`), which file to write (`outfile`), and the name of the final unfolded cause distribution (`nc_meas`).

```
[analysisbins]
bin = 0
stack = False
```

The `analysisbins` section tells the unfolder which bin(s) to consider for the unfolding. The `bin` option specifies the analysis bin number to access in the above ROOT files. The value must be an integer, and the unfolder expects the ROOT files to have the directory `bini` for the ith bin. For example, one could define a set of bins each with different data cuts, and thus require respectively different response functions. The `data inputfile` would also hold the extracted data distributions in their respective bins. Of course, we can only unfold a single **data** bin at a time. Thus, for most unfoldings, `bin` will be a single integer.

If one desires to stack several bins, for example to include response functions from two distinct cause populations, then the appropriate bin numbers are provided to `bin` via comma separated values. The `stack` flag also must be set to `True`. In this case, the `ne_meas` in the `data inputfile` **must** be located in the `bin0` directory, though this is not explicitly stated in the configuration file.

A quick note on stacked unfolding: there is nothing new in introducing a stacked unfolding. Looking back to Section 2.1, we see that the definition of the causes, C_μ, is completely general. Hence, one can simply 'stack' families of causes together. Visually, this is equivalent to setting two response matrices next to each other along the cause axis. For example, we can split the all-particle cosmic ray spectrum into two groups: light (P+He) and heavy (¿ He). Their respective response functions are generated separately (with the same analysis cuts) and placed in different bin directories of the same `stats_file`. A **single** effects distribution is extracted from the data, again with the same cuts, and must be in `bin0` of the `data inputfile`. When running the unfolder, the results for each stacked bin are saved in respective directories of the same `outfile`.

In general one must be mindful of the choice of initial priors for each bin when stacking. The simplest example is to stack the same response matrix with itself. If we choose priors for bins $0, 1$ with fixed ratio r for all cause bins, ie

$$\frac{P_0(C_\mu)}{P_1(C_\mu)} = r \quad \forall \mu. \tag{22}$$

then the unfolded cause distributions will retain this ratio through all iterations. Of course, summing the two distributions at any point in the unfolding gives the same result as using the single response matrix.

```
[prior]
# Options: Jeffreys or user defined f(x) such as 10*x**-1.6
# NB: comma separated funcs required for stacking...
func = Jeffreys
```

Here the user can define a functional form of the initial prior. The default is 'Jeffreys' prior [6]. A user provided function must be single-valued and conform to `numpy` syntax if special functions are used. If doing a stacked unfolding, formulae must be separated by commas for each bin.

```
[unfolder]
unfolder_name = Unf
max_iter = 100
# Smooth each iter w/Regulizer
smooth_with_reg = True
verbose = False
```

This section gives the user choice over the name of the unfolding function (`unfolder_name`), the maximum allowed number of iterations via `max_iter`, and whether to include regularization at each step with `smooth_with_reg`. Details of the regularization are found in the next section. The `verbose` flag prints out detailed information of the unfolding.

```
[regularization]
# Function form, in root format
reg_func = [0]*x**[1]
# Fit Range
reg_range = 1e4,1e10
# Give your parameters names!
param_names = Norm,Idx
# And initial values & limits
param_init = 1e8,-2.5
param_lim_lo = 1e2,-10
param_lim_hi = 1e20,-1
plot = False
verbose = False
```

The choice of regularization scheme can be manipulated in a manner consistent with ROOT histogram fitting. The functional form in `reg_func` is single-valued and the parameters must be labeled in sequential order via []. The restrictions on range in x is defined with `reg_range`. For example, when unfolding the all-particle spectrum, regularization to a power law is done on a subrange of interest in Energy. The fit parameters are given labels, initial values, and low and high limits per the respective keywords. Finally, for detailed fitting information, `plot` and `verbose` flags can be set appropriately.

```
[teststatistic]
# TS method. Options: rmd, chi2, pf, ks
ts_name = ks
# TS equivalence tolerance
ts_tolerance = 0.001
# TS cause range
ts_range = 1e4,1e10
verbose = True
```

The test statistic, TS, is used to compare the unfolded distributions from two consecutive iterations: $TS = TS(N_i, N_{i+1})$. This threshold to determine convergence is defined with the `ts_tolerance` keyword, and a subset of the cause range can be provided via `ts_range`. The test statistic functions available are the following:

- rmd: Maximum relative deviation, $\sup(\frac{|N_i - N_{i+1}|}{N_i + N_{i+1}})$
- chi2: Reduced χ^2
- pf: Bayes Factor (Pfendner Factor) [7]
- ks: Kolmogorov-Smirnov

```
[mixer]
mix_name = SrMixALot
# Cov Matrix. Options: DCM (faster), ACM (more correct)
error_type = ACM
```

The `mixer` section simply names the mixer function and has two methods for the calculation of the covariance matrix due to the limited statistics in the response function.

- DCM: D'Agositini method outlined in [1]
- ACM: Adye uncertainty from [4]; defined in this document with Eqs. 9 and 15 and fully outlined in Section 2.4

While the comment in this section claims that DCM is faster, the updated implementation of ACM no longer influences performance.

```
[flux]
# Function form, in root format
reg_func = [0]*x**[1]
# Fit Range
reg_range = 5e3,5e6
# Give your parameters names!
param_names = Norm,Idx
# And initial values & limits
param_init = 1e4,-2.5
param_lim_lo = 1e1,-5
param_lim_hi = 1e10,-1
plot = False
verbose = False
```

The final section is identical in form to the `regularization` section. However, it only pertains to fitting a differential flux once the unfolding procedure is finished. The parameters here are only read by the `FluxFit.py` script, and hence can be modified to test different fit parameters and/or functions after the unfolding.

References

[1] G. D'Agostini, “A Multidimensional unfolding method based on Bayes' theorem”, Nucl. Instrum. Meth. A **362** (1995) 487

[2] Z. Hampel-Arias, “Cosmic Ray Observations at the TeV Scale with the HAWC Observatory”, Ph.D. thesis, University of Wisconsin – Madison (2017)

[3] T. Adye, “Unfolding algorithms and tests using RooUnfold”, Proceedings of the PHYSTAT 2011 Workshop, CERN, Geneva, Switzerland, January 2011, CERN-2011-006, pp 313-318, arXiv:1105.1160

[4] T. Adye, “Corrected error calculation for iterative Bayesian unfolding”, Personal Website

[5] ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A **389** (1997) 81-86. See also http://root.cern.ch/

[6] Jeffreys, H., “An Invariant Form for the Prior Probability in Estimation Problems”. Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences. **186** (1007): 453–461

[7] BenZvi, S., Connolly, B., Pfendner, C.G., Westerhoff, S., “A Bayesian Approach to Comparing Cosmic Ray Energy Spectra”, The Astrophysical Journal **738** (2011) 1, arXiv:1106.1392