Fast Integration of Poisson Distributions for Dead Sensor Marginalization

Zoë Bilodeau[1,*], *Christina* Peters[2,3], and *Christopher* Tunnell[3,4]

[1]Skidmore College, Saratoga Springs, New York, USA.
[2]Department of Computer and Information Sciences, University of Delaware, Newark, Delaware, USA.
[3]Department of Physics and Astronomy, William Marsh Rice University, Houston, Texas, USA.
[4]Department of Computer Science, William Marsh Rice University, Houston, Texas, USA.

Abstract. In dual-phase time-projection chambers there are photosensor arrays arranged to allow for inference of the positions of interactions within the detector. If there is a gap in data left by a broken or saturated photosensors, the inference of the position is less precise and less accurate. As we are unable to repair or replace photosensors once the experiment has begun, we develop methods to estimate the missing signals. Our group is developing a probabilistic graphical model of the correlations between the number of photons detected by adjacent photosensors that represents the probability distribution over photons detected as a Poisson distribution. Determining the posterior probability distribution over a number of photons detected by a sensor then requires integration over a multivariate Poisson distribution, which is computationally intractable for high-dimensions. In this work, we present an approach to quickly calculate and integrate over a multidimensional Poisson distribution. Our approach uses `Zarr`, a `Python` array compression package, to manage large multi-dimensional arrays and approximates the log factorial to quickly calculate the Poisson distribution without overflow.

1 Introduction

1.1 XENONnT Experiment

This project was for the XENONnT experiment, a dark matter direct detection experiment located in Gran Sasso, Italy. It uses a dual-phase time projection chamber filled with xenon, which interacts with particles passing through the detector. The interactions produce photons which are detected by photosensor arrays on the top and bottom of the detector. The goal of the experiment is to use the data collected from the sensor arrays to determine what kind of particles interacted with the xenon.

1.2 Estimating Sensor Readings

The problem we aim to address in this work is related to malfunctioning sensors. The sensors are very sensitive and can break, also they cannot be fixed after the experimental apparatus is

*e-mail: zobilodeau@gmail.com

closed and the experiment is underway. In this work we study the correlations between the number of photons detected by adjacent sensors, which will allow us to estimate the number of photons a broken or malfunctioning sensor would have detected. The goal of estimating the missing sensor values is to improve the accuracy of positional reconstruction, without the need for special cases for gaps in sensor data. Accurate position reconstruction is necessary because backgrounds in XENONnT are position dependent. Near the metal walls of the xenon chamber there are many more particle interactions which originated from radioactive decay of the metal field shaping rings than those that originated from space.

In this work, we developed a `Python` package to quantify correlations between the number of photons detected by adjacent sensors in the particle detector and determine the most probable radial position of the interaction by marginalizing over all possible values of the broken sensors.

2 Multivariate Poisson Distribution

2.1 Poisson Distribution

The data we are looking at is the hit count of all PMT sensors on the top of the detector during a single event, and the sensors radial positions. This package needs to calculate both a multidimensional Poisson distribution when more than one PMT is dead, and a uni-variate Poisson for when just one sensor in the group is broken so that we can find the most probable number of photons a sensor would have detected regardless of how many sensors are broken. An event in this case is an interaction in the particle detector, and interactions occur independently of each other, and the number of photons detected by a sensor is discrete. A Poisson distribution is the most appropriate probabilistic distribution to describe the number of hits detected by a sensor.

2.2 Calculating a Multivariate Poisson Distribution

Our approach to calculating the multivariate Poisson distribution involves first calculating a uni-variate distribution for each broken sensor in a group of 7 sensors at each possible interaction position. The groups of sensors are hexagonal, one sensor in the middle and all the sensors that surround it, see Figure 1). This grouping pattern is so that sensors are close to each other in terms of radial position, which should find the strongest relationship between sensors' radial position relative to each other and the number of photons they detect. Figure 2 shows an example group of sensors for a simulated interaction.

The goal of the Poisson distribution is to be able to fill in the blank sensors with an estimate of the number of hits. If only one sensor is broken the distribution is complete. Otherwise, we need to multiply by the cross μ term,

$$P(k_0, k_1|\lambda_0, \lambda_1, \lambda_{0,1}) = e^{(-\lambda_0 - \lambda_1 - \lambda_{0,1})} \frac{{\lambda_0}^{k_0}}{k_0!} \frac{{\lambda_1}^{k_1}}{k_1!} \sum_{k=0}^{min(k_0,k_1)} \frac{k_1!}{k!(k_1-k)!} \frac{k_2!}{k!(k_2-k)!} k! \Big(\frac{\lambda_{0,1}}{\lambda_0 \lambda_1}\Big)^k \quad (1)$$

where k is a number of photons a sensor may have detected and λ is the mean number of photons likely detected in the range of possible k values. As the function contains the factorial of k, a value that can cause overflow due to its large size, we created an approximation of the cross μ term. Each uni-variate distribution is multiplied by its respective cross μ term to create a bi-variate Poisson distribution. The completed multivariate Poisson distribution is a large multidimensional array of probabilities from which we can find the estimated number of electrons detected by each broken sensor at each possible interaction position.

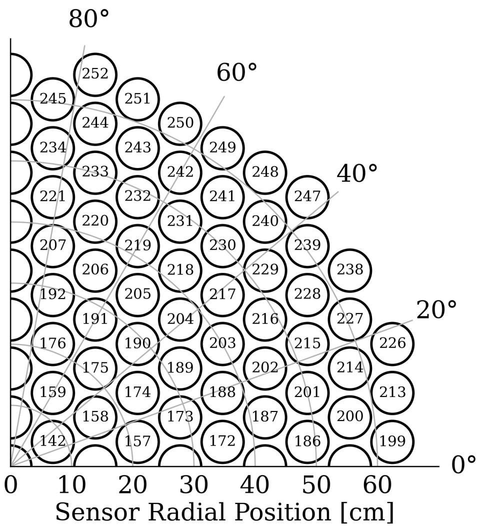

Figure 1. Arrangement of the sensors in the XENONnT detector. Credit: [1]

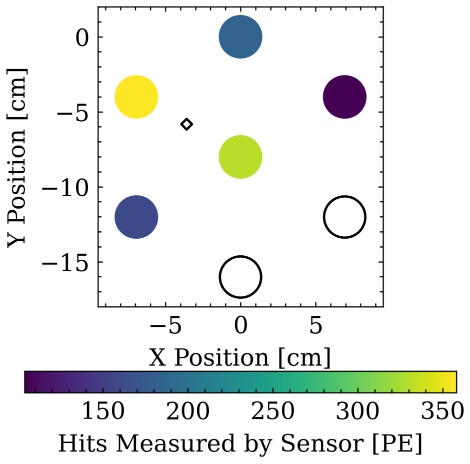

Figure 2. Example simulated hit pattern with broken sensors represented by empty circles, and the interaction position indicated by black diamond.

3 Modifying Multivariate Poisson Calculations

3.1 Split up Multivariate Calculations into Two Parts

To address the factorial issue, we broke the multivariate calculations into 2 parts. The first part is the PMF, or uni-variate Poisson:

$$P_u(k_0|\lambda_0) = e^{-\lambda_0}\frac{{\lambda_0}^{k_0}}{k_0!} \tag{2}$$

We modeled the PMF in this `Python` package after `Scipy`'s version [2]. The most efficient way to avoid overflow in this equation was to compute the log-PMF, as this makes it possible to use a log-factorial approximation in place of the factorial in the denominator. Additionally, in the case that we need to create a joint Poisson distribution instead of a multi-variate distribution, the log-probabilities can be added together to create the distribution instead of multiplied. This can prevent underflow from small probabilities being multiplied together and preserve the accuracy of the data. We tested a number of log-gamma and log-factorial approximations for accuracy and speed, and found that the Ramanujan log-factorial approximation,

$$\sqrt{\pi}\left(\frac{n}{e}\right)^n \sqrt[6]{8n^3 + 4n^2 + n + \frac{1}{30}} \tag{3}$$

ran the fastest of those we tried while maintaining a very high level of accuracy. It also scales well in terms of run-time, as it is affected less by input size than the Python factorial function.

If more than one sensor in the group is broken, then the multi-variate calculations are done to create a joint distribution. This joint distribution contains all possible combinations of probabilities across all broken sensors, as shown in Figure 3.

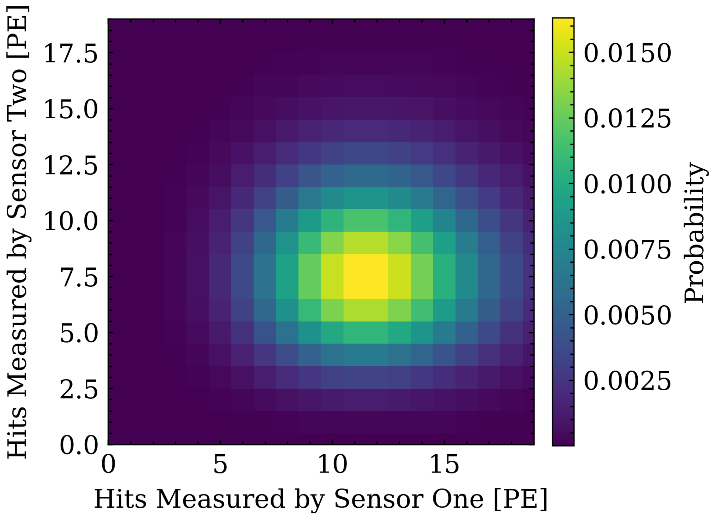

Figure 3. Joint Poisson Probability Distribution for 2 Broken Sensors. Each axis is the probability distribution of a certain sensor, where the color in the plot represents the probability that a certain number of photons was detected by that sensor summed with the probability of the other sensor at that point.

3.2 Function Approximations

To avoid computing factorials that would cause overflow, we found different function approximations. The first factorial we needed to address was in the Poisson probability mass function (uni-variate distribution). We worked with this function in two ways The Ramanujan log-factorial approximation ran the fastest of those we tried while maintaining a very high level of accuracy. After filling in a properly sized array with the univariate values, we then calculate the second part of the function, the cross mu term. The cross mu term also includes multiple factorials, as shown here:

$$\sum_{k=0}^{min(k_0,k_1)} \frac{k_1!}{k!(k_1-k)!} \frac{k_2!}{k!(k_2-k)!} k! \Big(\frac{\lambda_{0,1}}{\lambda_0 \lambda_1}\Big)^k \tag{4}$$

Unfortunately, due to the sum it is not possible to simply take the log of this section of the function, as summing log-probabilities will result in multiplied probabilities instead of added. Instead, there is an approximation which can handle these values without overflow:

$$e^{(i\ \pi\ min(k_0,k_1))} \Big(\frac{\lambda_{0,1}}{\lambda_0 \lambda_1}\Big)^{min(k_0,k_1)} U(-min(k_0,k_1), -min(k_0,k_1) + max(k_0,k_1) + 1, -\Big(\frac{\lambda_0 \lambda_1}{\lambda_{0,1}}\Big)) \tag{5}$$

where U is the confluent hypergeometric function, which we calculated using `Scipy` [2].

3.3 Calculating Distributions for Each Possible Interaction Position

For positional reconstruction, we are interested in determining the most likely radial position the interaction may have happened at. To find this, in addition to making a Poisson distribution for each broken sensor, there is a distribution for each broken sensor at each possible interaction position. The array of probabilities has an axis that corresponds to the possible interaction positions. The length of the axis is the number of possible interaction positions being considered, and the distribution for combination of broken sensor and interaction position is calculated with a unique μ value.

3.4 Marginalization

The final calculation is marginalizing over the number of photons axis, to find the most likely possible radial interaction position. This means simply summing over the number of photons axis, as our variables are discrete. For setting up joint distributions and integrating, `Numpy`'s broadcasting and summing methods are reliable due to their flexibility when working with variable numbers of axes (necessary due to variable number of broken sensors) and their speed; to calculate our final array all probabilities in the array are summed over the number of photons axis, meaning at the end of the function there is one probability at each possible interaction position [3].

4 Reducing Time and Storage Requirements

4.1 Storage Requirements

To reduce the amount of space required to store these distributions, we implemented the `Zarr` package. `Zarr` is a compression package for `Numpy` arrays, which works by dividing arrays into chunks which can be individually decompressed as needed. Values generated for the large joint distribution are stored in a `Zarr` array, and the only time the array is modified is during integration. To integrate using `Zarr`, the chunks being summed are decompressed, added together, and compressed again [4]. `Zarr` is effective for this program as the array is only modified in sections, meaning the whole array never needs to be decompressed.

4.2 Time Requirements

To improve the run-time, the `Scipy` stats package has a function for the Poisson probability mass function (PMF), it is made to be able to deal with special cases. Checking for special cases takes time, and as none of these cases apply to our data we are able to remove them entirely. The data being passed through these functions is the number of electrons detected, and that number will only ever be a positive integer. We wrote a simplified version of `Scipy`'s PMF with no special cases, and used `Numpy`'s vectorize function to further speed up PMF generation. Vectorize is able to apply a function to an entire `Numpy` array more quickly than a for-loop. As a large amount of the program is made up of loops and work with `Numpy` arrays, we used `Numba` decorators to improve the overall runtime [5].

5 Initial Results

At its current stage the package can only work with simulated data. When distributions get too large, underflow occurs and the probabilities are replaced with the minimum float that can be represented. If distributions are not too large, both the estimated values for sensors and the interaction position results appear plausible. The distribution at the most likely interaction position calculated from example data in Figure 4 is consistent with true number of hits simulated, shown as a black diamond.

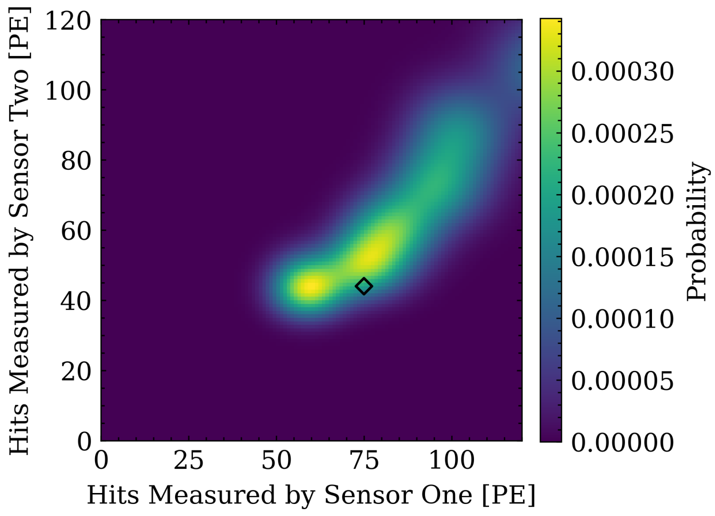

Figure 4. Joint probability distribution over photons detected by the broken sensors, both for the example simulated hit pattern shown in Figure 2. The black diamond indicates the true simulated value.

6 Future Work

The next step in this project is to investigate efficient methods of preventing underflow in large distributions. After this is resolved, we would do benchmark testing to see whether or not further runtime and storage improvements are necessary. If space is still an issue, we can work to further optimize `Zarr` compression by increasing the number of chunks created and test whether or not the chunks are efficient for integration. If the package is problematically slow, we can adjust our vectorized functions to use `Numba` vectorize, and potentially implement a lookup table when computing the Poisson distribution to avoid calculating common values many times.

References

[1] C. Peters, A. Higuera, S. Liang, V. Roy, W.U. Bajwa, H. Shatkay, C.D. Tunnell, arXiv e-prints arXiv:2205.10305 (2022), `2205.10305`

[2] P. Virtanen et al. (SciPy 1.0 Contributors), Nature Methods **17**, 261 (2020)

[3] C.R. Harris et al., Nature **585**, 357 (2020)

[4] A. Miles et al., *zarr-developers/zarr-python: v2.4.0* (2020), `https://doi.org/10.5281/zenodo.3773450`

[5] S.K. Lam, A. Pitrou, S. Seibert, *Numba: A llvm-based python jit compiler*, in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (2015), pp. 1–6