Summer Overview
Zoë Bilodeau

This summer I worked on writing a program for the XENON project, an experiment designed to detect dark matter or weakly interacting massive particles. The experiment takes place in a mountain in Gran Sasso Italy, and uses a duel-phase time projection chamber, which is a type of particle detector. It is filled with liquid and gaseous xenon with which particles passing through the detector interact. These interactions produce photons which are detected by arrays of sensors on the top and bottom of the detector. With the data we get from the particle detector, we are interested in understanding what interactions are occuring between the different particles to determine what kind of particles flew into the detector with the intention of identifying dark matter. One way we are doing this, and the one I worked on, is done by looking at the photons generated during collisions between the xenon and particles that are passing through the detector. Figuring out what correlations exist between the number of photons detected by each sensor in the particle detector will help us discern where the collision happened and what kind of interaction occurred between the particles. We are hoping these interactions will give off unique signals that will allow us to identify dark matter particles. Figuring out where in the tank the collision happened will help us to know when interactions may have occurred between particles from the metal walls of the tank produced in radioactive decay. If an interaction occurs close to the wall of the tank, we can't tell whether it was a space particle or a particle from the wall. In addition, this will help us to be able to deal with malfunctioning sensors. The sensors are very sensitive and are somewhat prone to breaking, but cannot be fixed without opening the detector. Finding the correlations between the number of photons detected by sensors based on position will allow us to estimate the number of photons the sensor likely detected.

To calculate the correlations we must create a multidimensional Poisson probability distribution over each sensor's radial positions in the array of sensors in groups of seven, and integrate over all possible photon intensities each broken sensor could have detected. Calculating these correlations is slow and costly; it requires a lot of small computations and storage. Since we are calculating these in groups of seven, we need to be able to handle up to 7 broken sensors, which means a 7-dimensional Poisson distribution, with the additional dimension for radial position. A 5 dimensional array of maximum pmf length would require about 40 terabytes of space, which is unreasonable for most computers. The correlations involve counting photons and therefore fit a Poisson probability distribution, the probability mass function (PMF) for which is $P(x) = \frac{e^{-\mu_*-\mu}}{x!}$. This function is problematic for computers due to the factorial, which causes overflow at x=170. Since we need to be able to calculate probabilities at x≈400, this function needed to be adjusted. The last problem I worked on was making the integration as quick as possible over a multidimensional probability distribution, as it can potentially take many many calculations. My job this summer was to write a program that can quickly and efficiently do this computation.

To address the issue of joint distribution space, we used the Python package zarr, a library developed to be able to divide numpy-like arrays into chunks and compress them. To prevent overflow when calculating the Poisson pmf we calculated the log Poisson PMF so that we would be able to use a log-factorial approximation that can handle x values up to ~400. We tried a number of log-factorial approximations, and found that Ramanujan's log-gamma approximation is one that scales well and runs quickly. Using the log Poisson PMF ended up being helpful in a different way since we could add probabilities together rather than multiply them together when making the joint distribution, which leaves less room for probabilities to get too small and round down to zero. For setting up joint distributions and integrating, Numpy's broadcasting and summing methods ended up being the best I tried due to their versatility and speed.