Explanation:

“vectorizing matrix multiplication along an axis without any for loops”

(x-µ) is a LARGE_NUMBER x 3 matrix and Σ^{-1} is 3x3 matrix. We can basically ignore the inverse part for demonstration purposes. Matrix multiplication is easy for a single row. $\Sigma^{-1} \cdot (\text{x-}\mu) = [1 \text{ x } 3] \cdot [3\text{x}3]$ However, we would need to loop over every row of (x-mu) to get the full results. This is too slow. A solution I found after trying a lot of things is to add a new axis to X and do the matrix multiplication in the newly created dimension!

Basic operation:

$$\Sigma^{-1}(\underline{x} - \underline{\mu})$$

Figure 1 Σ is a 3x3 matrix (*x*-µ) is distance to average RGB of a cluster for a large number of independent identically distributed samples

Here is the rough process:

*also outlined and tested in vectorized_dot_product_loop.py:

1. Add a new axis to X of depth Σ.rowsize. Now *X* is LARGE_NUMBER x h x 3 meaning there is are 3 copies of the original matrix in the new direction added.
2. Each “row” of *X* is now a 3x3 matrix which can do pointwise multiplication via numpy with Σ. This is the first step of matrix multiplication if it were done manually
3. Next, you take the sum along each row to column multiplication of x and sigma. This is the step that gives you the value of matrix multiplication for a row of the original *X.*
    a. One note is that numpy handles axes a little funny. In the example, I add an axes on the 2nd index and sum on the 1st index. But conceptually you are creating copies of the original row in a new dimension and multiplying them by their respective column of sigma and taking a sum all in the new dimension.

Take a look at the attached code if you want to play around with the idea. The pay off is that same operation over 100k rows goes from ~7s to ~0.4s when shifting from a for loop to the vectorized solution.

CONTEXT:

The scenario is implementing the expectation maximum algorithm for an unspecified number of dimensions and clusters. Basically K-Means but using a probability rather than assignment for k and an undefined number of clusters + data size. The important part for this use case is below. It is the equation for a multi-variate gaussian distribution for a sample value x where x is chosen from independent random samples.

$$\mathcal{N}(\underline{x}\ ;\ \underline{\mu}, \Sigma) = \frac{1}{(2\pi)^{d/2}} |\Sigma|^{-1/2} \exp\left\{ -\frac{1}{2}(\underline{x} - \underline{\mu})^T \Sigma^{-1} (\underline{x} - \underline{\mu}) \right\}$$

For the specific project, you have a pixel (x) with RGB values. Each cluster over a range of pixels is defined by an average (μ) of R,G, and B intensity as well as a covariance (Σ) between each value. Notably, a single pixel is 1X3. The μ vector (average) is also 1X3 and the covariance is 3x3. This all works out nicely in the equation above and you end up with a single float value which is the probability a pixel is in the cluster defined by μ and Σ. The problem arises when you want probability for a large number of pixels. X becomes LARGE_NUMBER X 3 and the natural option is to loop over the large number to get your vector of probabilities. This is where the NumPy trick comes in that I think is pretty neat. After a bit of vectorizing, everything is easily moved to a single calculation for all pixels and a single cluster except for the matrix multiplication of sigma (the covariance matrix) and x-mu (distance of a pixel from the average RGB for a cluster). That is what is addressed via vectorizing.