Image Reconstruction with Cimmino's Algorithm

Kate Suraev (s224854029)

September 15, 2025

In this project, we explore the implementation of Cimmino's algorithm for image reconstruction problems, such as those encountered in computed tomography (CT). The algorithm is particularly useful for solving large-scale linear systems that arise in these applications due to its ability to handle inconsistent systems and its parallelisable nature. We do not aim to provide a perfectly reconstructed scan but rather to demonstrate the possibility of using Cimmino's algorithm in a parallel computing environment to achieve a reasonable approximation of the original image.

0.1 Cimmino's Algorithm Overview

To give a high-level overview, Cimmino's algorithm is a row-iterative algorithm that simultaneously reflects the current approximation point (estimate) across all hyperplanes defined by the equations of the linear system. The subsequent approximation is then computed as the average of these reflections. The result is a convergent sequence of approximations that approaches either the exact solution in the case of a consistent system or the least-squares solution in the case of an inconsistent system.

Given an $m \times n$ matrix A with rows a_i, a vector $b \in \mathbb{R}^m$ and positive weights ω_i for $i = 1, 2, \ldots, m$, the algorithm can be expressed as follows:

$$x^{k+1} = \sum_{i=1}^{m} \frac{\omega_i}{\omega} y^{(k,i)} = x^k + 2 \sum_{i=1}^{m} \frac{\omega_i}{\omega} \frac{b_i - a_i^T x^k}{\|a_i\|^2} a_i \quad \text{for } k = 0, 1, 2, \ldots \ \textbf{cimmino} \tag{1}$$

Here, x^k is the current approximation, $y^{(k,i)}$ is the reflection of x^k across the hyperplane defined by the i-th equation, ω_i are positive weights assigned to each equation, and $\omega = \sum_{i=1}^{m} \omega_i$.

In simpler terms, the algorithm can be expressed in matrix form as:

$$x^{k+1} = x^k + \frac{2}{\omega} A^T D^T D (b - A x^k) \quad \text{for } k = 0, 1, 2, \ldots \tag{2}$$

where D is a diagonal matrix with entries $D_{ii} = \sqrt{\omega_i} / \|a_i\|$.

$$D = \begin{bmatrix} \frac{\sqrt{\omega_i}}{\|a_1\|} & 0 & \cdots & 0 \\ 0 & \frac{\sqrt{\omega_2}}{\|a_2\|} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\sqrt{\omega_m}}{\|a_m\|} \end{bmatrix} \tag{3}$$

For simplicity, we choose particular weights $\omega_i = \|a_i\|^2$ for $i = 1, 2, \ldots, m$. This choice simplifies the diagonal matrix D to the identity matrix I, leading to a more straightforward update rule:

$$x^{k+1} = x^k + \frac{2}{\omega} A^T (b - A x^k) \quad \text{for } k = 0, 1, 2, \ldots \tag{4}$$

0.2 Implementation Details

We intend to perform a simulation of a CT scan by setting the geometry parameters of our ‘scanner’, generating a projection matrix that models the scanner geometry, performing a scan of a given image to obtain a sinogram (a vector of measurements), and then using Cimmino’s algorithm to reconstruct the original image from the sinogram.

Phantom: A phantom image is a standard test image used in medical imaging to simulate the human body **NIST**. In this project, we use the Shepp-Logan phantom, a widely used test image in CT imaging that models a head cross-section. This is our ground truth image that we aim to reconstruct. The 256x256 Shepp-Logan phantom is generated in Python using Astra Toolbox **astrashepp**.

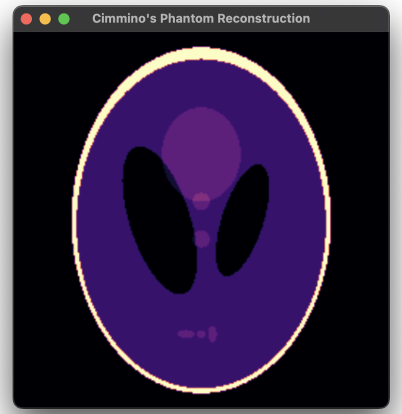

Figure 1: Shepp-Logan Phantom (256x256)

Sinogram: A 2D array where each row represents the projection of the image at a specific angle **sinogram**. This is our measurement data obtained from the simulated CT scan and can be thought of as b in the linear system $Ax = b$. The sinogram is generated in a Metal kernel using the phantom image and a ray marching algorithm. As seen below, the sinogram does not look like the original image, but it contains all the information needed to reconstruct it.

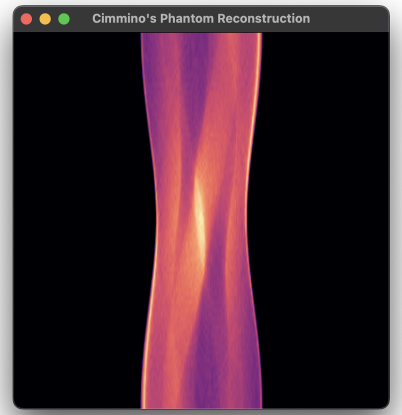

Figure 2: Sinogram (90 angles, 725 detectors)

We also use metal kernel functions to normalise the sinogram values to a 0-1 range for better visualisation and computation. For this, we use a SIMD parallel reduction algorithm, inspired by an example from the MSL specification, to find the maximum value in the sinogram and then normalise all values accordingly. Without normalisation, we still get a good reconstruction, but the convergence is slower. Additionally, we are not able to properly visualise the sinogram as an image since the values are too large.

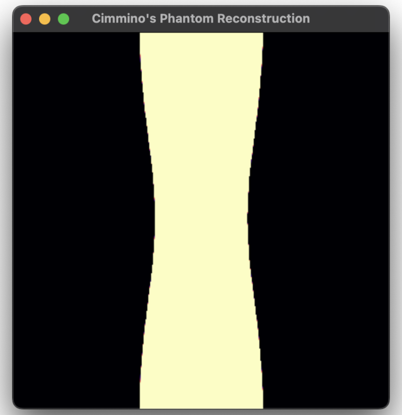

Figure 3: Not normalised Sinogram (90 angles, 725 detectors)

Projection Matrix: A matrix that transforms the image space into the sinogram space based on the scanner geometry **projectionmatrix**. This matrix represents the system of equations we need to solve to reconstruct the image. In other words, it is A in the linear system $Ax = b$. The projection matrix is generated using Astra Toolbox in Python, which provides efficient methods for creating projection matrices based on specified scanner geometries **astraprojection**. This is then saved to a binary file as a sparse matrix in the Compressed Sparse Row (CSR) format to optimise memory usage and read into the Metal application.

Scanner Geometry: The configuration of the CT scanner, including the number of angles, the number of detectors, the detector spacing and total angle degree **scanner_geom**. This defines how the projection matrix is constructed and influences the quality of the reconstruction. In this project, we set the scanner geometry parameters as follows:

- Number of angles: 90 (i.e., projections taken every 2 degrees over 180 degrees)
- Number of detectors: $\lceil \sqrt{2} * 256 \rceil = 725$ (to ensure full coverage of the image width)
- Detector spacing: 1 unit
- Total angle degree: 180 degrees

It's important to note, that the more angles and detectors we have, the better the reconstruction quality likely will be, but this also increases the computational load and memory requirements. For example, using the parameters we have chosen (90 angles and 725 detectors based on a 256x256 image) which are rather modest, the projection matrix becomes quite large (256*256 x 90*725 = 65536 x 65250), with 18737864 non-zero elements and $\approx$ 4.28 billion total elements if stored as a dense matrix. This is why we opted to use sparse matrix representation and parallel processing to handle the computations efficiently.

Cimmino's Algorithm Implementation: The core of the project is the implementation of Cimmino's algorithm in a parallel computing environment using Metal. The algorithm iteratively updates the image estimate by reflecting it across the hyperplanes defined by each equation in the linear system and averaging these reflections. The algorithm implementation is decomposed into two steps/kernels:

- **Reconstruction Kernel:** Each thread is responsible for one row of the projection matrix. It computes the dot product of the current estimate and the corresponding row of the projection matrix, subtracts this from the corresponding measurement in the sinogram to obtain the residual, and then computes that ray's contribution to the reflection. These results are stored in an update buffer.
- **Update Kernel:** This kernel reads the update buffer and adds the contributions from all rays to the current estimate, effectively averaging the reflections to produce the new estimate. Each thread is responsible for one pixel in the image.

Results

All programs are tested with the same parameters: 90 angles, 725 detectors and 256x256 image size and use the same projection matrix generated with Astra Toolbox. The sinogram produced in the Metal application is saved to a .bin file and loaded into other applications for comparison. We are only comparing the reconstruction time (the reconstruction loop) to have a fair comparison between the different implementations - sequential, OpenMP and Metal. The time taken to generate the sinogram, load the projection matrix and render the images is not included in the comparison.