Christopher Sweeney
12/06/24
CPSC 4387

Building a Distributed Solver for Linear Equations

Introduction

Linear equations form the backbone of many sophisticated technologies. All linear equations model relationships that scale proportionally, so it is a useful branch of applied mathematics for human-scale systems. Systems of linear equations, or linear systems, can be used to collect and track multifaceted data, and the data structures within the linear systems are likewise useful as abstractions and containers to move and manipulate said data. A linear equation typically takes the form of ax+by+c=0 in two dimensions, representing a line in a Cartesian plane. Linear systems further expand upon these basic lines, allowing multiple equations to solve for multiple unknowns simultaneously.

Many physical processes follow this pattern of proportional scale, and software processes make use of linear systems on many levels of abstraction, not least of which is the *array.* With the advent of large datasets and complex modeling, solving linear systems can be a computational bottleneck. The motivation behind distributed solutions lies in leveraging parallelism to reduce computational time and manage large-scale problems that cannot fit into a single machine's memory.

This solver is a fun approach to understanding distributed linear equation-solving. Implemented in Unity, using NetCode for GameObjects, it allows multiple clients to cooperatively solve a system of equations while visualizing their progress.

Properties of Linear Equations

A linear equation is defined as an algebraic equation in which each term is either a constant or the product of a constant and a single variable. General forms include:

- Standard form**:** Ax+By+C=0
- Slope-intercept form**:** y=mx+b

These equations can extend to multiple variables, leading to systems of linear equations that are typically represented in matrix form. These problems are in contrast to higher and lower order functions, like exponential or radical functions, that do not scale proportionally with input. Key properties include:

Linearity**:** The relationship between variables is proportional.
Solution Uniqueness**:** Systems can have one solution, no solution, or infinitely many solutions. The resulting matrices take characteristic forms that give clues as to the solution.
Superposition Principle**:** In linear systems, the sum of two solutions is also a solution. Combinations of vectors in the linear system thus form subspaces that can be reasoned about.

Linear equations involving multiple unknowns can be represented as:

$$Ax = b$$

Where A is a two-dimensional matrix of coefficients, x is a vector of unknowns, or observations, and b is a vector of constants. Solving a problem of this form involves building an augmented matrix, specified by A|b, and using it to solve for x. This augmented matrix form is the shape the game board takes in my solver.

Matrix representation allows for manual solving through computational methods, and this notation has been developed for more than two centuries to allow for powerful and creative functions to be realized by linear systems. Some approaches require heavy calculation, extending beyond Clients can use the information provided to solve for x however they like, and there are a lot of approaches that might work:

Substitution is done by solving one variable in terms of another, substituting in subsequent equations.
Elimination requires manipulating equations to eliminate variables sequentially.
Gaussian Elimination involves row reductions to achieve an upper triangular matrix.
LU Decomposition factors matrix A into lower/upper triangular matrices (LU) for easier calculation.
Iterative Approaches:

- Jacobi Method: Updates each variable using values from the previous iteration.
- Kaczmarz Method – Projects possible solutions on constituent rows until convergence.
- Gauss-Seidel Method: Uses the most recently updated values for convergence.

A linear system must be well-defined to be solvable, and often times the system can have many solutions or no solution at all. In some cases, regression analysis can find the closest vector to a theoretical solution, and often the system must be optimized to find the "best" solution vector. There are important properties of some matrices that bound the possible solution spaces and make the system easier to compute, and they are helpful to predict if algorithmic iterations will converge on a solution. Convergence is ensured under conditions like diagonal dominance (elements above or below the diagonal are mostly zero) or when the matrix is positive definite (all variables must be positive, and often whole numbers).

Pros and Cons of Distributed Linear Solutions

Dealing with linear systems requires some specific capabilities. Because they can reach arbitrary sizes, a system must have enough memory to store the problem while solving and working on it. The computational complexity is not necessarily high, but the system is so large that it can take quite a while to solve, even if the calculations are rudimentary. And because the algorithms to solve large linear systems are iterative, rounding errors and memory problems can be exaggerated over time.

One strategy of mitigating these problems is to parallelize the process of solving large linear systems. Distributed solutions divide the problem among several computational devices, each responsible for a subset of calculations. This allows for scaling resources and experimenting with modular approaches. There are some problems with distributing a system as such, because memory and computation constraints are replaced with communication overhead and increased system complexity. Increased complexity can lead to convergence challenges, so a robust algorithm must be used that is fault tolerant and capable of handling high bandwidth networking.

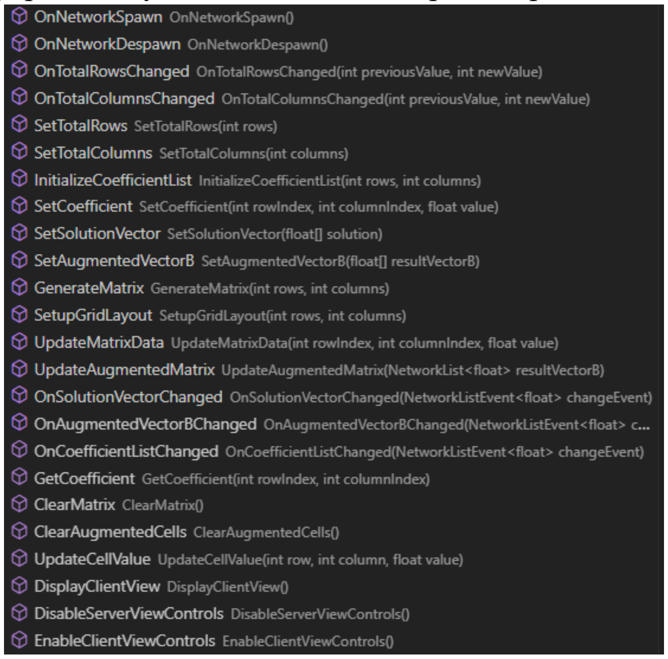
There exists many projects and technologies that leverage this sort of distribution to solve and handle large linear systems. Telecommunication stations, data centers, and software companies must handle a lot of customer data and real-time communications, and these linear systems underpin the transmission/reception of data packets, error correction, and inference of user location that are required for all wired and wireless communications. Scientific simulation and forecasting is performed with collections of linear systems that model certain physical or chemical behaviors, and these models are scaled and combined to research complex environmental conditions.

Most machine learning and neural networking techniques rely on techniques suited for linear systems. Regression analysis and principle component analysis are key solutions in statistics that were developed by mathematicians studying linear systems, and neural networks handle layered multidimensional arrays for token embedding and system weights.

Game Flow: MatrixVisualizer, ServerControl and ClientControl Scripts

MatrixVisualizer - Game board setup and state storage

The MatrixVisualizer script is a Network Object that holds the matrix information passed by the server so that clients can share a common game board and reason towards the same victory conditions. The matrix is built in a Grid Layout component that is populated by Prefabs that hold respective parts of the standard form equation, Ax = b. The server submits size information, the number of rows and columns, and this is used to build a list of coefficients list that is the length of their product.



```
OnNetworkSpawn OnNetworkSpawn()
OnNetworkDespawn OnNetworkDespawn()
OnTotalRowsChanged OnTotalRowsChanged(int previousValue, int newValue)
OnTotalColumnsChanged OnTotalColumnsChanged(int previousValue, int newValue)
SetTotalRows SetTotalRows(int rows)
SetTotalColumns SetTotalColumns(int columns)
InitializeCoefficientList InitializeCoefficientList(int rows, int columns)
SetCoefficient SetCoefficient(int rowIndex, int columnIndex, float value)
SetSolutionVector SetSolutionVector(float[] solution)
SetAugmentedVectorB SetAugmentedVectorB(float[] resultVectorB)
GenerateMatrix GenerateMatrix(int rows, int columns)
SetupGridLayout SetupGridLayout(int rows, int columns)
UpdateMatrixData UpdateMatrixData(int rowIndex, int columnIndex, float value)
UpdateAugmentedMatrix UpdateAugmentedMatrix(NetworkList<float> resultVectorB)
OnSolutionVectorChanged OnSolutionVectorChanged(NetworkListEvent<float> changeEvent)
OnAugmentedVectorBChanged OnAugmentedVectorBChanged(NetworkListEvent<float> c...
OnCoefficientListChanged OnCoefficientListChanged(NetworkListEvent<float> changeEvent)
GetCoefficient GetCoefficient(int rowIndex, int columnIndex)
ClearMatrix ClearMatrix()
ClearAugmentedCells ClearAugmentedCells()
UpdateCellValue UpdateCellValue(int row, int column, float value)
DisplayClientView DisplayClientView()
DisableServerViewControls DisableServerViewControls()
EnableClientViewControls EnableClientViewControls()
```

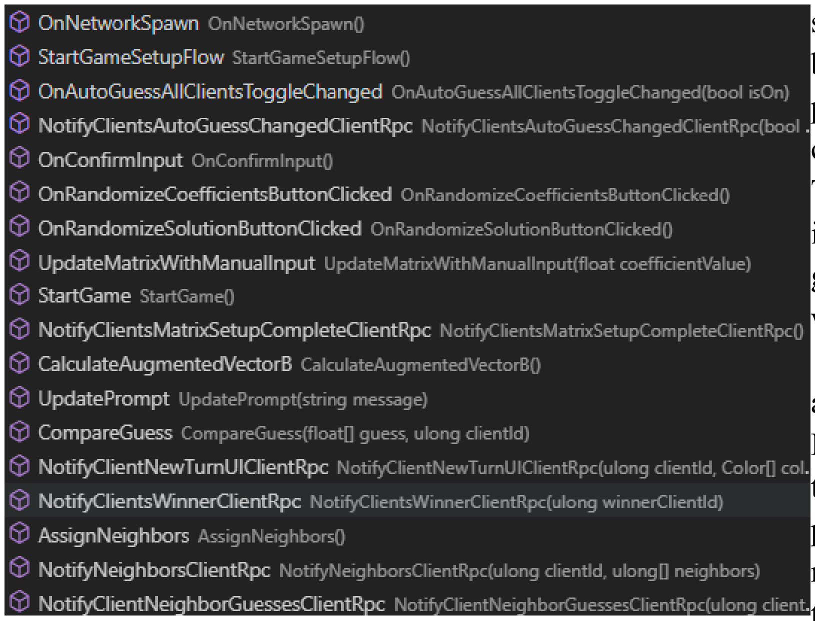
When the server is done inputting information, the this script puts the Prefabs inside the grid layouts such that the linear system is in augmented matrix form, a commonly established format to solve for x. When a client spawns, it uses the MatrixVisualizer to build an identical matrix to that shown on the server, and this script hides the server-specific UI from the client.

Only one attribute in this script is private: the solutionVector that clients are trying to guess. It is used to compute the augmentedSolutionB vector, but ServerControl handles all logic to compare client guesses for accuracy. This script also holds the current game state so that both server and clients have access to it but neither “owns” it.

ServerControl - Game Initialization and Flow

The ServerControl script starts with the OnNetworkSpawn() method, setting up the game board dimensions and allowing the server to enter coefficients. A single input field and confirm button are the bulk of the UI, but buttons like randomizeCoefficientsButton simplify testing by auto-filling matrix values. This is especially helpful for large matrices, saving time it would take to fill in elements one at a time.

Communication with clients is orchestrated through Remote Procedure Calls (RPCs), such as NotifyClientsMatrixSetupCompleteClientRpc(). When key actions occur, such as when the matrix setup is complete or when results need to be displayed, these calls pass some parameters to and tell the appropriate clients that it should call a certain method. These calls are used to pass neighbor information, give feedback for client guesses, and trigger the game board setup when server input is finalized.



```
OnNetworkSpawn OnNetworkSpawn()
StartGameSetupFlow StartGameSetupFlow()
OnAutoGuessAllClientsToggleChanged OnAutoGuessAllClientsToggleChanged(bool isOn)
NotifyClientsAutoGuessChangedClientRpc NotifyClientsAutoGuessChangedClientRpc(bool
OnConfirmInput OnConfirmInput()
OnRandomizeCoefficientsButtonClicked OnRandomizeCoefficientsButtonClicked()
OnRandomizeSolutionButtonClicked OnRandomizeSolutionButtonClicked()
UpdateMatrixWithManualInput UpdateMatrixWithManualInput(float coefficientValue)
StartGame StartGame()
NotifyClientsMatrixSetupCompleteClientRpc NotifyClientsMatrixSetupCompleteClientRpc()
CalculateAugmentedVectorB CalculateAugmentedVectorB()
UpdatePrompt UpdatePrompt(string message)
CompareGuess CompareGuess(float[] guess, ulong clientId)
NotifyClientNewTurnUIClientRpc NotifyClientNewTurnUIClientRpc(ulong clientId, Color[] col
NotifyClientsWinnerClientRpc NotifyClientsWinnerClientRpc(ulong winnerClientId)
AssignNeighbors AssignNeighbors()
NotifyNeighborsClientRpc NotifyNeighborsClientRpc(ulong clientId, ulong[] neighbors)
NotifyClientNeighborGuessesClientRpc NotifyClientNeighborGuessesClientRpc(ulong client
```

The game architecture is built around the server being a "Host", so all RPC calls to clients are also executed by the instance running as a "Server". To prevent this unwanted operation, all methods triggered by a ClientRPC from the server have an if statement gating the server from executing said method.

ClientControl - Client Initialization and Guessing by Sliders

Finally, the ClientControl script calls MatrixVisualizer to spawn the player UI, initializes slider elements that represent the guess values for each unknown, and call a ServerRPC to send the guess to the client. The ActivateSliders() method presents each player with interactive sliders and input fields, and these slider values are combined to form a guess. The server or client can choose to "autoguess", which automatically adjusts the sliders to algorithmically converge on an answer. This is great for demo purposes for when clients are to be run autonomously, en masse. Instead of listing the methods in ClientControl, let's focus on a few methods from the ServerControl and ClientControl scripts that results in a sort of game loop that happens until the problem is solved.

CompareGuess() from ServerControl and NewTurnUI() from ClientControl

When a player submits their guess using the slider, the SubmitGuessServerRpc() method sends this data to the server's CompareGuess() method for evaluation. The feedback from this method is sent to the client, where the UI is updated using NewTurnUI() to reflect how close the player's guess was, using a color-coded system to guide player actions.

Algorithmic AutoGuess Functionality

The ApplyAutoGuess() method leverages the projection parameter and error correction to iteratively move slider values towards the solution, until convergence is reached. This dual convergence method was necessary beyond the algorithm outlined in the midterm project report, because the algorithm offered improved convergence rates only if the clients were near-optimally converging by themselves. Without the error correction and resulting gradient-optimization, the clients would only converge to a common answer, not a correct one. With both methods, they converge to a common answer, but this average guess also converges towards the real answer at the same time.

Neighbor Guessing and Synchronization

NetCode for GameObjects handles the networking layer. MatrixVisualizer and ServerControl scripts are attached to corresponding GameObjects, but client guesses are passed around as float arrays, parameters in RPC calls to and from the server. Therefore, the server acts as the arbiter of who is who's neighbor and updates the list of neighbor's guesses for each client. This is a straightforward approach to handling the client guesses, but fails to meaningfully describe the network topology to any extent. This falls short of ambitions in the midterm project report, as the nearest neighbors graph was supposed to add some visualization for the server to follow while the clients submitted their guesses.

One approach might be to build different methods for different network topology and let the server choose in the same way it chooses the matrix dimensions. Another might be to allow for graphically arranging clients on a grid and connecting them arbitrarily, such that the graph can be parsed by a method no matter the structure. In any case, this capability is not realized, but the average neighbor guess is simple to calculate in the AutoGuess() method after receiving the float array.

Conclusion

This report reviewed the implementation of a game – the distributed linear equation solver. We explored linear algebra concepts, solution methodologies, and the benefits of a distributed approach, and then the basic structure and flow of the program was elucidated. Future iterations could include dynamic addition and collaboration by clients, with inclusion of a nearest neighbor graph to monitor convergence. Pre-designed templates for network topology might allow teachers to setup a visual environment to show students the effects of network topology on message propagation and teamwork. The project has such applications in education, by helping students understand iterative convergence methods, and could extend to collaborative problem-solving.

Further, the game could include external sources of linear data such that it becomes a visual tool for simulation and cooperation. A group of people can view sensor readings and interventions together, and could contribute individualized strategies for solving computational problems that can be visualized and aggregated into a common understanding of a problem and its solution space.

This report presents a complete execution of the basic version 1.0 of this program. It succeeds in displaying both computational mathematics and a basic distributed system in an interactive format.

Works Cited

Mou, S., Lio, J., & Morse, A. S. (10/03/2024). A distributed algorithm for solving a linear algebraic equation. Retrieved from https://arxiv.org/pdf/1503.00808

Olfati-Saber, R., Fax, J. A., & Murray, R. M. (2007). Consensus and cooperation in networked multi-agent systems. Retrieved from https://labs.engineering.asu.edu/acs/wp-content/uploads/sites/33/2016/09/Consensus-and-Cooperation-in-Networked-Multi-Agent-Systems-2007.pdf

Stack Overflow. (n.d.). Computational intensity of transposing a matrix vs. calculating the inverse. Retrieved November 27, 2024, from https://stackoverflow.com/questions/7446389/computational-intensity-of-transposing-a-matrix-vs-calculting-the-inverse

Unity Technologies. "Messaging System." *Unity Multiplayer Documentation*. Accessed November 25, 2024. https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/messaging-system/

Unity Technologies. "Slider (UI Toolkit)." *Unity Manual*. Accessed November 12, 2024. https://docs.unity3d.com/Manual/UIE-uxml-element-Slider.html

Unity Technologies. "NetworkList<T>." *Unity Netcode for GameObjects API*. Accessed November 24, 2024. https://docs.unity3d.com/Packages/com.unity.netcode.gameobjects@1.0/api/Unity.Netcode.NetworkList-1.html