A Distributed Solver for Linear Equations

By: Christopher Sweeney

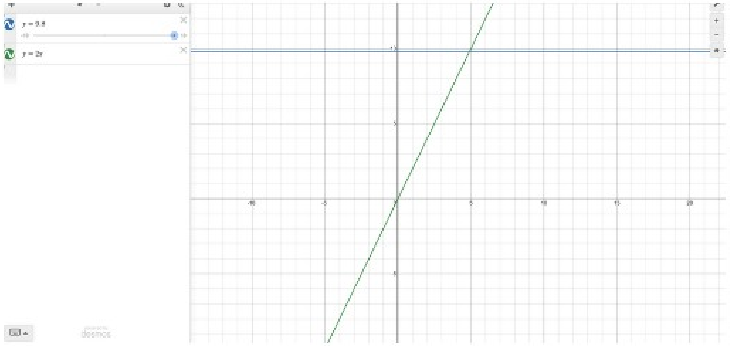
12/06/24

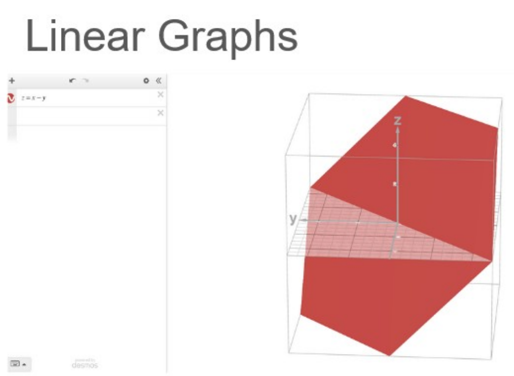
A History of Linear Equations

- Gottfried Leibniz (1646-1716) - German polymath of mathematics, philosophy, science, and politics
- Arranged the coefficients of a system of linear equations to find the solution(s) -
- Later developed by German mathematician Gauss (1777-1855)
- in the West - Japanese mathematician Seki Takakazu (1642-1708) developed it in the East

What are Linear Equations?

- A system of equations with variable of degree one
- Acts a linear mapping, or transformation
- Presented in the form Ax=b
  - A in the linear system
  - x is an observed vector space
  - b is the resulting vector space after linear transformation
- Contrasts with radical, quadratic, cubic, and higher/lower order functions
- Applications in various fields of research, computing and machines





Linear Systems in Action

- Transformations, isomorphisms
- Beamforming, error correction
- Massive MIMO
- Time-series data
- Production monitoring, reporting
- Token embeddings, weights

- Computer vision, graphics
- Wireless networking (5G, IoT)
- Radio arrays
- HF trading, Folding@Home
- Smart manufacturing/cities
- LLMs and neural networks

Common Algorithms to Solve Linear Equations

- Square Matrices (m=n)
  - Gaussian Elimination (Chen)
  - LU Decomposition
- Under-determined (m<n)
  - Pseudoinverse
  - Minimum Norm (O(n^{2-3}) !!!
  - Kaczmarz's Algorithm (Stack)
- Over-determined (m>n)
  - Least Squares
  - SVD
- Sparse
  - Sparse LU
  - Conjugate Gradient
- Ill-conditioned
  - Regularization
  - SVD

Solving with a Distributed Algorithm

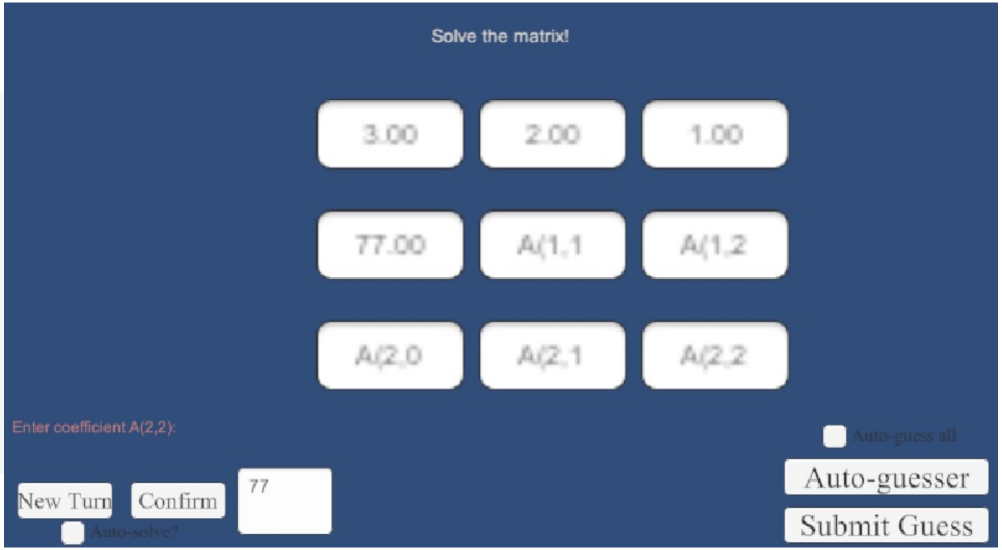
- Distributing workload allows for:
  - Larger linear systems, scalability
  - Parallelism
- Increased capabilities, with trade-offs:
  - Bandwidth and latency
  - Fault tolerance
  - Higher initial costs, except when hardware limited (Oltafi-Saber)

Solving with a Distributed Algorithm

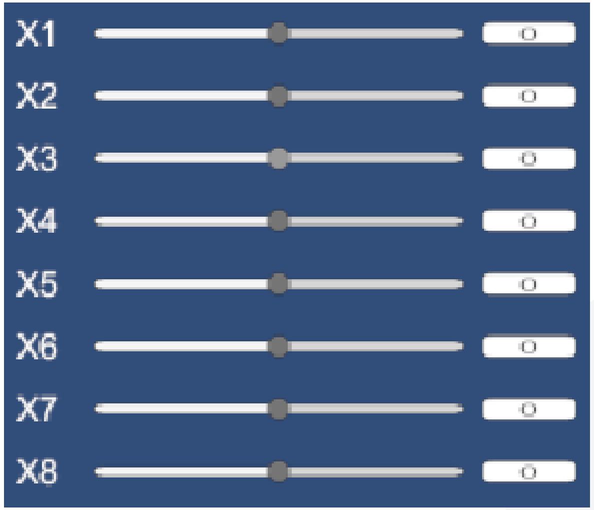
- Server sets up a linear system for clients to guess the solution
- Clients send guesses, receive feedback and neighbors’ guesses
- Clients race to and quickly converge on solution, finishing the “game”

Program Control Flow

- Server sets up A and x, calculates b

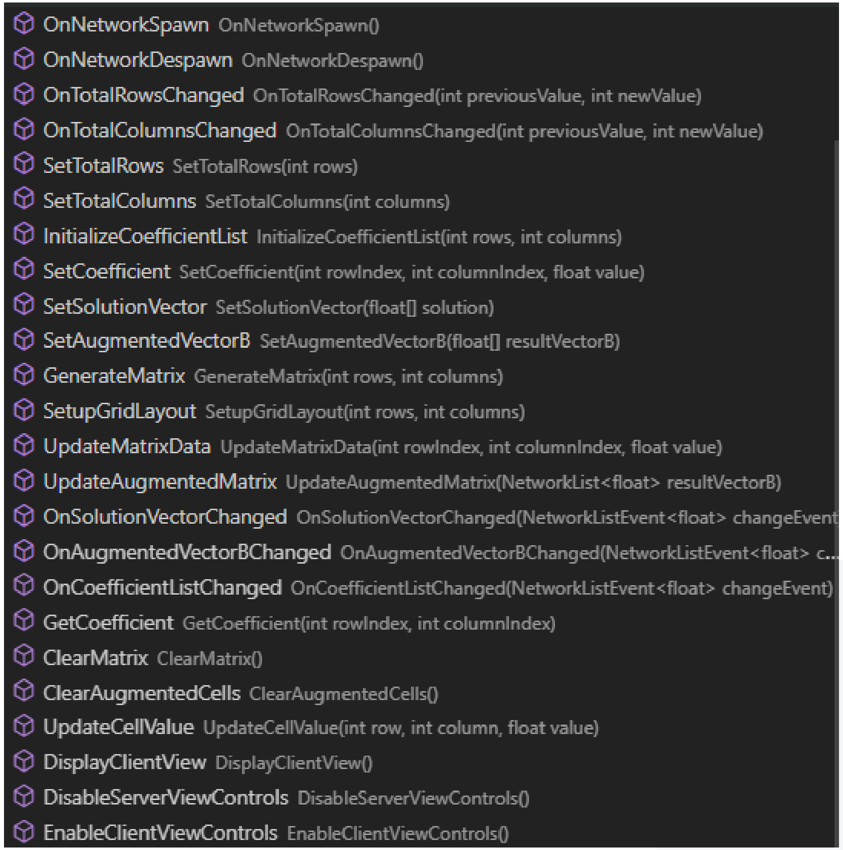


- Clients guess by moving sliders and send to server



Matrix Network Object

- Handles “game board”
  - Shared between server and all clients
  - Attributes public except for the NetworkList<float> solutionVector



Server Setup

- NotifyClientsAutoGuessChangedClientRpc
- NotifyClientsMatrixSetupCompleteClientRpc
- NotifyClientNewTurnUIClientRpc
- NotifyClientsWinnerClientRpc
- NotifyNeighborsClientRpc
- NotifyClientNeighborGuessesClientRpc

```
[ClientRpc]
1 reference
private void NotifyClientNewTurnUIClientRpc(ulong clientId, Color[] colorVector, float[] guess, float[] errorVector, bool[] solvedFlags)
{
    if (NetworkManager.Singleton.LocalClientId == clientId)
    {
        FindObjectOfType<ClientControl>()?.NewTurnUI(colorVector, guess, errorVector, solvedFlags);
    }
}
```

Client Networking

- ClientCanvas Network Object
- ClientRPCs in ClientControl script:

- OnMatrixSetupComplete()
- NewTurnUI()
- OnWinnerAnnounced()
- SetNeighbors()
- UpdateNeighborGuesses()

- ServerRPCs in ClientControl script:

```
[ServerRpc(RequireOwnership = false)]
    4 references
    private void SubmitGuessServerRpc(float[] guess, ServerRpcParams rpcParams = default)
    {
        // Call server logic to compare guess
        Debug.Log($"Received guess from client: {rpcParams.Receive.SenderClientId}");
        FindObjectOfType<ServerControl>().CompareGuess(guess, rpcParams.Receive.SenderClientId);
    }
```

Adapting an Optimized Algorithm to use with Unity

- The chosen method is based on work from Mou, Lio, and Morse:

Client's guess

$$x_i(t+1) = x_i(t) - \frac{1}{m_i(t)} P_i \left(m_i(t) x_i(t) - \sum_{j \in \mathcal{N}_i(t)} x_j(t) \right), \quad t \geq 1$$

Client's guess

Projection parameter

Neighbor's guess

Number of neighbors

```csharp
private void AutoGuess()
{
    // Calculate the new guess based on the algorithm.
    float sumNeighborGuesses = 0f;
    foreach (float guess in neighborGuesses)
    {
        sumNeighborGuesses += guess;
    }

    if (numberOfNeighbors > 0)
    {
        float m_i = numberOfNeighbors;
        float P_i = 1.0f; // Simplified projection value. Adjust depending on the specifics of P_i.
        float newGuess = currentGuess - (1 / m_i) * P_i * (m_i * currentGuess - sumNeighborGuesses);

        currentGuess = newGuess;
        SendGuessToServer(currentGuess);
        Debug.Log("AutoGuess: " + currentGuess);

        //UpdateBoundingBox();
    }
}
```

```csharp
public void ApplyAutoGuess(float[] averageGuess, float[] errorVector)
{
    if (matrixVisualizer.currentState == MatrixVisualizer.GameState.AdjustingSliders && lastAverageGuess != null)
    {
        // Retrieve the current values of the sliders (the current guess vector)
        float[] currentValues = GetGuessVector();
        int dimension = currentValues.Length;

        iterationStep++; // Increment step count for each iteration to track algorithm progression

        // Parameters for learning and convergence
        float initialLearningRate = 0.1f; // Controls how quickly the guesses converge to the solution
        float fixedStepSize = 1f; // Forces an incremental change to ensure convergence proceeds step by step

        // Calculate the difference vector between the current values and the average guess
        float[] differenceVector = new float[dimension];
        for (int i = 0; i < dimension; i++)
        {
            differenceVector[i] = averageGuess[i] - currentValues[i];
        }

        // Calculate the dot product between the current guess vector and the difference vector
        float dotProduct = 0f;
        for (int i = 0; i < dimension; i++)
        {
            dotProduct += currentValues[i] * differenceVector[i];
        }

        // Use the dot product to influence the projection scaling
        float projectionScalingFactor = projectionParameter * dotProduct;

        // Update each value in the guess vector based on the projection scaling factor, difference, and fixed step size
        for (int i = 0; i < dimension; i++)
        {
            // Skip any sliders that have already been solved
            if (isSolved[i])
            {
                continue;
            }

            // Ensure error remains positive and doesn't approach zero (avoiding division by zero)
            float error = Mathf.Abs(errorVector[i]);
            error = Mathf.Max(error, 0.001f); // Set a minimum value for error to prevent instability

            // Calculate the adaptive scaling factor based on the current error
            float adaptiveScalingFactor = initialLearningRate / (1 + error); // As error decreases, step size becomes smaller

            // Adjust the current value to move towards the average guess using adaptive scaling
            currentValues[i] += adaptiveScalingFactor * differenceVector[i];

            // Apply the projection scaling factor to the current value
            currentValues[i] += projectionScalingFactor * differenceVector[i];
```

Dual Algorithmic Convergence

- Distributed algorithm relies on differentiated clients
- Solution known, so cost function is used
- Gradient descent optimization + signal averaging (Chong 231)
- Faster server would speed up former
- More/faster clients would speed up latter

Distributed Difficulties

- Network Objects require serialization, some types need custom
- Host/Server ambiguity and lots of gates
- Handing control flow to clients and back to server can get stuck
- Slider behavior is erratic, disturbs algorithm
  - Lots of listeners and clamping
- Neighbor assignment easy, neighbor guess handling is hard

Extending the Solver

- Building nearest neighbor graph to visualize algorithm
- Allow real-time matrix manipulation, just like sliders
- Neighbor strategies and network topology templates
- Piping in linear system – home automation, PID, wireless
- Matrix randomization, schemas, lessons

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

Chen, X. (n.d.). Kaczmarz Algorithm, row action methods, and statistical learning algorithms. Iowa State University. https://faculty.sites.iastate.edu/esweber/files/inline-files/KaczmarzSGDrevised.pdf

Chong, E. K. P., & Zak, S. H. (2013). An introduction to optimization (4th ed.). Wiley.