Db2-interp

A Console ABD Block-Tridiagonal Solver with Proofs and Db2 Option

Tyler Jung

2025

Overview

This project is a pure Node.js console application that demonstrates:

- Object-Oriented Programming concepts: inheritance, aggregation, composition.
- Data structures: a custom singly linked list with iterative and recursive reversal.
- Proof techniques: using hypotheses and invariants to argue correctness.
- Counting principles: permutations and combinations, with code helpers.

It solves Almost Block Diagonal (ABD) block-tridiagonal systems, inspired by numerical analysis methods for boundary-value problems. Optionally, it can fetch numeric series from a Db2 database to populate the right-hand side vector.

Why this exists

Instead of teaching each application how to solve ABD systems, this single service:

- Encapsulates the math in a reusable module.
- Provides consistent, tested solutions with clear proofs of correctness.
- Can serve as a middleman between *any application* and Db2.

How it works

1. The user runs the console program with either:
   - Built-in demo mode.
   - Db2 mode (fetches a series of data points).
   - Counting mode (`--count n r`).
2. In ABD solve mode, the system is constructed as:

$$\begin{bmatrix} A_0 & B_0 & 0 & \cdots & 0 \\ C_1 & A_1 & B_1 & \cdots & 0 \\ 0 & C_2 & A_2 & \cdots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & & C_{p-1} & A_{p-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{p-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{p-1} \end{bmatrix}.$$

3. A Thomas-style block algorithm does forward elimination, then back-substitution.

Proofs of Correctness

We present hypotheses and invariants in a conversational but rigorous way.

Forward Elimination

Hypothesis. Eliminating block C_i using information from row $i-1$ preserves equivalence of the system.
Invariant. At the start of step i, all rows $0, \ldots, i-1$ are in upper block-triangular form.
Update rule.

$$T = A_{i-1}^{-1} B_{i-1}, \qquad y = A_{i-1}^{-1} d_{i-1}$$
$$A_i \leftarrow A_i - C_i T, \qquad d_i \leftarrow d_i - C_i y$$

Why it holds. This is exactly the Gaussian elimination step: we cancel C_i while maintaining equality. The invariant ensures previous rows stay solved.

Back Substitution

Hypothesis. Once all C_i are removed, the system is upper block-triangular and solvable backwards.
Invariant. When solving for x_i, all x_j for $j > i$ are already correct.
Update rule.

$$x_{p-1} = A_{p-1}^{-1} d_{p-1}, \qquad x_i = A_i^{-1} \left(d_i - B_i x_{i+1}\right).$$

Why it holds. Each row depends only on its own block and the next unknown. Induction guarantees correctness as we move backwards.

Linked List Reverse

Iterative.
Invariant: the processed prefix is fully reversed, the suffix untouched. Each step flips one pointer until done.
Recursive. Hypothesis: reversing the tail first then appending the head reverses the whole list. Base case: list of length 0 or 1 is already reversed.

Counting Principles

Permutations.

$$nP_r = \frac{n!}{(n-r)!}.$$

Invariant: at each pick, the number of choices decreases by one.
Combinations.

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}.$$

Reasoning: permutations divided by $r!$ reorderings removes duplicates.

CS Concepts Demonstrated

- **Aggregation & Composition:** The pipeline aggregates tasks and composes a linked list.
- **Inheritance:** `Algorithm` base class extended by `ABDSolverLite`.
- **Linked Lists:** Both iterative and recursive reversal.
- **Counting Principles:** Helper functions for permutations and combinations.
- **Proof Techniques:** Hypothesis + invariant reasoning across math and data structures.

Competency Map

- **OOP and Data Structures:** inheritance, aggregation, linked lists.
- **Proofs:** friendly but rigorous correctness arguments.
- **Counting:** implemented and applied to batching/partitioning reasoning.