1D Random Walk with PETSc: Data Structures and Implementation Documentation

Random Walk Project

October 13, 2025

Contents

1 Introduction

This document provides comprehensive documentation for the 1D random walk implementation using PETSc (Portable, Extensible Toolkit for Scientific Computation). The code demonstrates a simple random walker on a periodic 1D grid using PETSc's DMDA (Distributed Mesh Data Array) framework.

2 Data Structures

2.1 PETSc Data Structures

Data Structure	Type	Purpose
`DM da`	DMDA Object	Manages 1D structured grid with periodic boundaries
`Vec walker_density`	PETSc Vector	Stores walker position as density distribution
`PetscErrorCode ierr`	Error Code	Handles PETSc function return codes

Table 1: PETSc Data Structures

2.1.1 DM (Data Management) Object

- **Variable**: `DM da`
- **Creation**: `DMDACreate1d(PETSC_COMM_SELF, DM_BOUNDARY_PERIODIC, grid_size, 1, 0, NULL, &da)`
- **Purpose**:
  - Manages the 1D structured grid topology
  - Handles periodic boundary conditions automatically
  - Provides framework for domain decomposition (scalable to parallel)
  - Serves as foundation for vector creation and grid operations
- **Parameters**:
  - `PETSC_COMM_SELF`: Single process communicator
  - `DM_BOUNDARY_PERIODIC`: Periodic boundary conditions
  - `grid_size`: Number of grid points (100)
  - `1`: Degrees of freedom per grid point
  - `0`: Stencil width (not used for this application)

2.1.2 Vec (Vector) Object

- **Variable**: `Vec walker_density`
- **Creation**: `DMCreateGlobalVector(da, &walker_density)`
- **Purpose**:
  - Represents the probability density of walker position
  - Stores value 1.0 at walker's current position, 0.0 elsewhere
  - Enables visualization of walker state on the grid

- Compatible with DMDA structure for future extensions

- **Operations Used**:
  - `VecZeroEntries()`: Initialize all values to zero
  - `VecSetValue()`: Set walker position (clear old, set new)
  - `VecAssemblyBegin/End()`: Finalize vector modifications
  - `VecView()`: Display vector contents

2.2 Standard C++ Data Structures

Data Structure	Type	Purpose
`grid_size`	`PetscInt`	Number of grid points (100)
`num_steps`	`PetscInt`	Total simulation steps (1000)
`walker_position`	`PetscInt`	Current walker grid index
`gen`	`std::mt19937`	Mersenne Twister random number generator
`step_choice`	`std::uniform_int_distribution<>`	Uniform distribution for step direction

Table 2: Standard C++ Data Structures

2.2.1 Simulation Parameters

- **Grid Size**: `PetscInt grid_size = 100`
  - Defines the size of the 1D periodic domain
  - Walker moves on indices $[0, 99]$ with wraparound

- **Number of Steps**: `PetscInt num_steps = 1000`
  - Total number of random walk steps to simulate
  - Each step moves walker by ± 1 grid unit

- **Walker Position**: `PetscInt walker_position`
  - Tracks current grid index of the walker
  - Initialized to middle of domain: `grid_size / 2`
  - Updated each step with periodic boundary conditions

2.2.2 Random Number Generation

- **Generator**: `std::mt19937 gen(rd())`
  - High-quality Mersenne Twister pseudorandom generator
  - Seeded with `std::random_device` for non-deterministic initialization

- **Distribution**: `std::uniform_int_distribution<> step_choice(0, 1)`
  - Generates uniform integers: 0 or 1
  - Mapped to step directions: $0 \to -1$, $1 \to +1$
  - Ensures equal probability for left/right movement

3 Algorithm Overview

3.1 Initialization Phase

1. Initialize PETSc environment
2. Create 1D DMDA with periodic boundaries
3. Create global vector for walker density
4. Set initial walker position at domain center
5. Initialize random number generator

3.2 Simulation Loop

For each time step:

1. Clear current position in density vector
2. Generate random step direction (± 1)
3. Update walker position with periodic boundary conditions
4. Set new position in density vector
5. Assemble vector changes
6. Output progress (every 100 steps)

3.3 Cleanup Phase

1. Display final walker position and grid state
2. Destroy PETSc vectors and DM objects
3. Finalize PETSc environment

4 Scalability Design

The current implementation uses single-process execution (`PETSC_COMM_SELF`) for simplicity, but the DMDA framework provides a clear path for scaling:

- **Parallel Processing**: Change to `PETSC_COMM_WORLD`
- **Multi-dimensional**: Extend to 2D/3D with `DMDACreate2d/3d`
- **Multiple Walkers**: Increase degrees of freedom per grid point
- **Complex Physics**: Add finite difference stencils and time stepping

5 Future Extensions

- HDF5 output using `PetscViewerHDF5`
- Trajectory tracking with vector snapshots
- Multiple walker simulations
- 2D/3D random walks
- Parallel processing capabilities
- Advanced boundary conditions