Quantum Dot Simulation Suite Documentation

AI Software Developer

May 13, 2025

Contents

1 Introduction

This document provides an overview of the physics and code structure for a suite of quantum dot simulators. These simulators solve the Schrödinger and Poisson equations self-consistently to model the behavior of electrons in semiconductor quantum dot devices. The suite includes tools for 1D and 2D simulations, as well as specialized simulations for pinch-off characteristics, charge stability diagrams, and Coulomb diamonds. The simulations are designed to mimic experiments performed on gated semiconductor nanostructures.

2 Semiconductor Quantum Dot Devices

Semiconductor quantum dots (QDs) are nanoscale structures capable of confining electrons (or holes) in all three spatial dimensions. Due to this strong confinement, their energy levels become discrete, similar to those of an atom, leading to them often being referred to as "artificial atoms".

2.1 Formation in Semiconductor Heterostructures

A common method for creating QDs is by using patterned metallic gates on the surface of a semiconductor heterostructure. A typical structure involves growing a layer of a wider bandgap semiconductor (like AlGaAs) on top of a narrower bandgap semiconductor (like GaAs). Electrons from donor atoms in the AlGaAs layer transfer to the GaAs, forming a thin layer of highly mobile electrons at the interface, known as a two-dimensional electron gas (2DEG). This 2DEG is confined in the growth direction (typically z) by the bandgap difference.

Metallic gates deposited on the surface above the 2DEG can deplete the underlying electron gas when a negative voltage is applied. By carefully designing the shape and voltage of multiple gates, the 2DEG can be locally depleted, pinching off channels and isolating small puddles of electrons. These isolated puddles form the quantum dots, confined laterally (in x and y) by the electrostatic potential created by the gates.

Figure 1 illustrates a schematic cross-section of such a device structure.

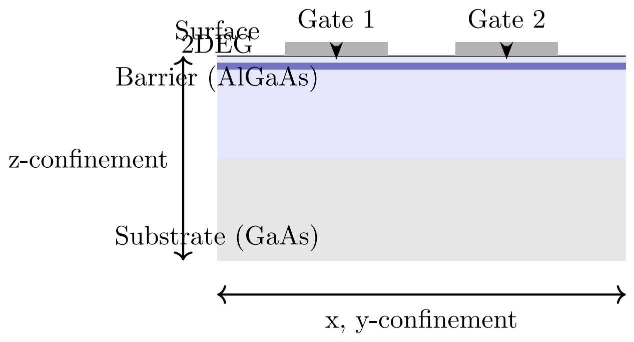

Figure 1: Schematic cross-section of a gated semiconductor quantum dot device. Top gates deplete the 2DEG, creating lateral confinement.

2.2 Key Properties: Confinement and Coulomb Blockade

The strong confinement in QDs leads to several key properties:

- **Quantum Confinement**: The energy levels for electrons in the dot become discrete, separated by energy gaps much larger than the thermal energy at low temperatures. Electrons can only occupy these specific energy states.
- **Coulomb Blockade**: Adding an extra electron to a quantum dot requires overcoming the electrostatic repulsion from the electrons already present. This costs an additional energy, the charging energy (E_C). This energy cost leads to a "Coulomb blockade" of current flow when the dot is connected to leads, unless the applied voltage is sufficient to overcome E_C.

These properties make QDs promising candidates for qubits in quantum computing and for studying fundamental quantum mechanics.

2.3 Simulated Device Geometry: Double Quantum Dot

The simulation suite focuses on a common and important device structure: the double quantum dot (DQD). A DQD consists of two quantum dots coupled together, typically formed by an arrangement of plunger gates (P1, P2) that define the potential wells for each dot, and barrier gates (B1, B2, B3) that control the coupling between the dots and to external leads (not explicitly simulated here, but implied).

The scripts model the gates using 2D (or 1D in `simulate_1d_dot.py`) Gaussian potential profiles, where the amplitude is proportional to the applied gate voltage. This provides a smooth, realistic potential landscape. Figure 2 shows a top-down schematic of the gate layout used in the 2D simulations.

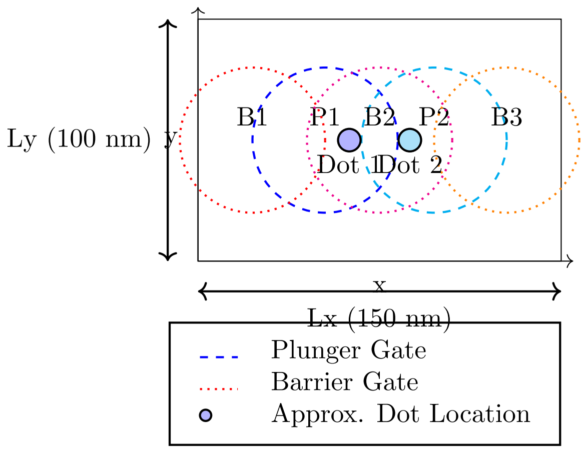

Figure 2: Top-down schematic of the gate layout for the simulated double quantum dot. Plunger gates (P1, P2) define the dots, while barrier gates (B1, B2, B3) control coupling.

By simulating the electronic properties of this DQD structure under varying gate voltages, the suite can reproduce characteristic experimental results like pinch-off curves and charge stability diagrams.

3 Physics Background

The core of the simulation lies in the self-consistent solution of the Schrödinger and Poisson equations.

3.1 Schrödinger Equation

The time-independent Schrödinger equation describes the quantum mechanical states of electrons in a given potential. For a single electron with effective mass m_{eff} in a potential $V(\mathbf{r})$, the equation is:

$$\left[-\frac{\hbar^2}{2m_{\text{eff}}}\nabla^2 + V(\mathbf{r})\right]\psi_i(\mathbf{r}) = E_i\psi_i(\mathbf{r}) \tag{1}$$

where $\hbar$ is the reduced Planck constant, $\psi_i(\mathbf{r})$ are the eigenfunctions (wavefunctions), and E_i are the corresponding eigenvalues (energy levels). The total potential $V(\mathbf{r})$ is a sum of the external potential $V_{\text{ext}}(\mathbf{r})$ (due to gates, as discussed in Section 2) and the electrostatic potential $\phi(\mathbf{r})$ due to the electron charge itself: $V(\mathbf{r}) = V_{\text{ext}}(\mathbf{r}) - e\phi(\mathbf{r})$.

3.2 Poisson Equation

The Poisson equation relates the electrostatic potential $\phi(\mathbf{r})$ to the charge density $\rho(\mathbf{r})$:

$$\nabla^2\phi(\mathbf{r}) = -\frac{\rho(\mathbf{r})}{\epsilon} \tag{2}$$

where $\epsilon = \epsilon_r\epsilon_0$ is the permittivity of the material (ϵ_r is the relative permittivity and ϵ_0 is the vacuum permittivity). The charge density $\rho(\mathbf{r})$ is calculated from the occupied electron states:

$$\rho(\mathbf{r}) = -e\sum_i f(E_i, E_F, T)|\psi_i(\mathbf{r})|^2 \tag{3}$$

where e is the elementary charge, and $f(E_i, E_F, T)$ is the Fermi-Dirac distribution function, which gives the probability of occupation for an energy state E_i at a given Fermi level E_F and temperature T. A factor of 2 is included for spin degeneracy. In the current implementation, a zero-temperature approximation ($T \to 0$) is often used, where $f(E_i, E_F, 0) = 1$ if $E_i < E_F$ and 0 otherwise.

3.3 Self-Consistent Solution

The Schrödinger and Poisson equations are coupled: the potential in the Schrödinger equation depends on the charge density (via the Poisson equation), and the charge density in the Poisson equation depends on the wavefunctions (via the Schrödinger equation). This necessitates a self-consistent solution:

1. **Initial Guess**: Start with an initial guess for the electrostatic potential $\phi(\mathbf{r})$ (e.g., $\phi(\mathbf{r}) = 0$ or the converged potential from a nearby gate voltage point for sweep simulations).
2. **Total Potential**: Calculate the total potential $V(\mathbf{r}) = V_{\text{ext}}(\mathbf{r}) - e\phi(\mathbf{r})$.
3. **Solve Schrödinger Equation**: Solve the Schrödinger equation using $V(\mathbf{r})$ to find eigenvalues E_i and eigenvectors $\psi_i(\mathbf{r})$.
4. **Calculate Charge Density**: Calculate the new charge density $\rho_{\text{new}}(\mathbf{r})$ using the computed E_i and $\psi_i(\mathbf{r})$ and the Fermi-Dirac distribution.
5. **Solve Poisson Equation**: Solve the Poisson equation with $\rho_{\text{new}}(\mathbf{r})$ to obtain a new electrostatic potential $\phi_{\text{new}}(\mathbf{r})$.
6. **Check Convergence**: Compare $\phi_{\text{new}}(\mathbf{r})$ with the previous $\phi(\mathbf{r})$. If the difference (e.g., measured by a norm) is below a specified tolerance, the solution is converged.
7. **Mix and Iterate**: If not converged, mix the new and old potentials (e.g., $\phi(\mathbf{r}) = (1 - \alpha)\phi(\mathbf{r}) + \alpha\phi_{\text{new}}(\mathbf{r})$, where α is a mixing parameter, typically between 0 and 1) and return to step 2. Mixing is crucial for numerical stability. Adaptive mixing schemes can also be employed.

This iterative process is repeated until convergence is achieved or a maximum number of iterations is reached.

4 Code Documentation

The simulation suite consists of several Python scripts, each tailored for specific types of simulations. They share common physical constants, material parameters, and core solver functions.

4.1 Common Elements

4.1.1 Physical Constants and Material Parameters

All scripts define:

- **Physical Constants**: $\hbar$ (hbar), m_e (electron mass), e (elementary charge), ϵ_0 (vacuum permittivity).
- **Material Parameters (GaAs)**: m_{eff} (effective mass, $0.067 \cdot m_e$), ϵ_r (relative permittivity, 12.9).

4.1.2 Simulation Grid

A 1D or 2D numerical grid is defined:

- `L` (1D) or `Lx`, `Ly` (2D): Length of the simulation domain.
- `N` (1D) or `Nx`, `Ny` (2D): Number of grid points.
- `x`, `y`: Arrays of grid point coordinates.
- `dx`, `dy`: Grid spacing.

For 2D simulations, `X` and `Y` meshgrids are created using `np.meshgrid` with `indexing='ij'` to match matrix indexing conventions.

4.1.3 Core Functions

`get_external_potential(X, Y, voltages)` Calculates the external potential profile based on gate voltages. Uses Gaussian profiles for each gate's influence, as described in Section 2.3. The `voltages` argument is a dictionary mapping gate names (e.g., "P1", "B2") to their applied voltage values.

`solve_schrodinger(potential)` (1D) or `solve_schrodinger_2d(potential_2d)`] Solves the time-independent Schrödinger equation using a finite difference method on a sparse matrix representation of the Hamiltonian. It finds the lowest few eigenvalues and corresponding eigenvectors using `scipy.sparse.linalg.eigsh`.

`calculate_charge_density(eigenvalues, eigenvectors, fermi_level)` (1D) or `calculate_charge_density_2d(eig` `eigenvectors_2d, fermi_level)`] Calculates the electron charge density using the computed eigenvalues, eigenvectors, and the specified Fermi level, applying the Fermi-Dirac distribution (often in the zero-temperature limit).

`solve_poisson(charge_density)` (1D) or `solve_poisson_2d(charge_density_2d)`] Solves the Poisson equation using a finite difference method with Dirichlet boundary conditions (potential fixed at boundaries, typically to zero). It constructs a sparse matrix for the Laplacian operator and solves the linear system using `scipy.sparse.linalg.spsolve` or an iterative solver like GMRES as a fallback.

`solve_poisson_2d_spectral(charge_density_2d)` (2D only) An alternative Poisson solver using spectral methods (Fast Fourier Transforms - FFTs). This method is efficient but inherently assumes periodic boundary conditions, which may not be physically appropriate for all device simulations compared to Dirichlet conditions.

`self_consistent_solver(voltages, fermi_level, ...)` (1D) or `self_consistent_solver_2d(voltages, fermi_level, ...)`] Implements the self-consistent iteration loop described in Section 3.3. It takes applied gate voltages and the Fermi level as input and iteratively calls the Schrödinger and Poisson solvers, mixing the potential until convergence is reached. It returns the converged total potential, charge density, eigenvalues, and eigenvectors (or None if convergence fails). The 2D version allows selecting the Poisson solver type.

4.2 `simulate_1d_dot.py`

This script performs a 1D Schrödinger-Poisson simulation for a quantum dot device.

- **Purpose**: Simulates a 1D quantum dot, typically representing a cross-section or a simplified model of a double dot structure defined by plunger and barrier gates.
- **Key Features**:
  - Defines a 1D grid.
  - `get_external_potential`: Models gates using 1D Gaussian potentials.
  - Solves 1D Schrödinger and Poisson equations.
  - Performs self-consistent calculation.
  - Plots potential profiles, charge density, and wavefunctions.

4.3 `simulate_2d_dot.py`

This script performs a standard 2D Schrödinger-Poisson simulation for a fixed set of gate voltages.

- **Purpose**: Simulates a 2D quantum dot, allowing for more realistic device geometries and potential landscapes. It's useful for visualizing the electron distribution and energy levels for a specific gate configuration.
- **Key Features**:
  - Defines a 2D grid (`Nx`, `Ny`).
  - `get_external_potential`: Models gates using 2D Gaussian potentials.

  - `solve_schrodinger_2d`: Solves the 2D Schrödinger equation using a 5-point finite difference stencil.
  - `solve_poisson_2d`: Solves the 2D Poisson equation using finite differences with Dirichlet boundary conditions.
  - `solve_poisson_2d_spectral`: An alternative spectral Poisson solver is available but not used by default in the main execution block.
  - `self_consistent_solver_2d`: Manages the 2D self-consistent loop. Allows selection of Poisson solver type.
  - Plots 2D contour maps of total potential, external potential, charge density, and the ground state probability density. Includes visualization of gate positions as ellipses.

4.4 `simulate_pinchoff.py`

This script simulates the pinch-off characteristics of a quantum dot device by sweeping a gate voltage.

- **Purpose**: To observe how the potential barrier under a gate changes as its voltage is swept, leading to channel pinch-off and depletion of electrons from the dot.
- **Key Features**:
  - Based on the 2D simulation framework.
  - Sweeps the voltage of a specified gate (e.g., "B2") over a defined range.
  - For each voltage point in the sweep:
    - Runs the 2D self-consistent solver.
    - Estimates the minimum potential under the swept gate (barrier height) by sampling a slice of the potential.
    - Calculates the total number of electrons in the simulation domain by integrating the charge density.
  - Plots:
    - Pinch-off curve: Minimum barrier potential vs. swept gate voltage.
    - Number of electrons vs. swept gate voltage.
    - Final potential landscape for the last voltage point simulated.
  - Allows selection between finite difference and spectral Poisson solvers for the self-consistent loop.

4.5 `simulate_charge_stability.py`

This script simulates charge stability diagrams by sweeping two gate voltages.

- **Purpose**: To map out regions of stable integer electron numbers in the quantum dot system as two plunger gate voltages are varied. This reveals the characteristic honeycomb pattern of charge stability, a signature of Coulomb blockade.
- **Key Features**:
  - Based on the 2D simulation framework, often with a reduced grid size (`Nx`, `Ny`) compared to `simulate_2d_dot.py` for faster computation over many voltage points.
  - Sweeps two specified gate voltages (e.g., "P1", "P2") over 2D ranges.
  - For each pair of (V_{Gate1}, V_{Gate2}) values:
    - Runs the 2D self-consistent solver.
    - Calculates the total number of electrons in the system by integrating the charge density using `calculate_total_electrons`.
  - Implements sweep strategies:
    - `row_by_row`: Standard raster scan.

    * `hilbert`: Sweeps points along a Hilbert space-filling curve using the `hilbertcurve` library. This strategy can improve convergence by using the converged potential from a nearby point in parameter space as a warm start (`initial_potential_V`) for the current point's self-consistent calculation. The `get_hilbert_order` function generates this sequence.
  - Stores the converged electrostatic potential from the previous point (in the chosen sweep order) to use as an initial guess (warm start) for the next point, potentially speeding up convergence, especially for the Hilbert sweep.
  - Stores the total number of electrons for each voltage pair in a 2D map.
  - Plots:
    * A 2D color map of the total number of electrons as a function of the two swept gate voltages.
    * An additional plot with rounded electron numbers to emphasize the integer plateaus and transitions.
  - Saves the raw data (voltages, electron map) to a '.npz' file and plots to image files in an output directory.
  - Allows selection between finite difference and spectral Poisson solvers.

4.6 `simulate_coulomb_diamonds.py`

This script simulates Coulomb diamonds, which are closely related to charge stability diagrams.

- **Purpose**: Similar to charge stability, this script maps out electron number as a function of two gate voltages, typically used to identify Coulomb blockade regions and extract parameters like charging energy.
- **Key Features**:
  - Based on the 2D simulation framework.
  - Sweeps two specified gate voltages (e.g., "P1", "P2").
  - For each voltage pair, runs the self-consistent solver and calculates the total number of electrons.
  - Plots a 2D color map of the electron number, forming the characteristic Coulomb diamond pattern.
  - Uses a zero-temperature approximation for charge density calculation.
  - Note: This script is a simplified version compared to `simulate_charge_stability.py` and does not include features like warm starts or different sweep strategies.

5 Numerical Methods

5.1 Finite Difference Method

The Schrödinger and Poisson equations are discretized using the finite difference method. For example, the second derivative $\frac{d^2\psi}{dx^2}$ is approximated as:

$$\frac{d^2\psi}{dx^2} \approx \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{dx^2} \tag{4}$$

where ψ_i is the value of ψ at grid point x_i, and dx is the grid spacing. In 2D, the Laplacian $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is approximated using a 5-point stencil. This transforms the differential equations into a system of linear algebraic equations, which can be represented by sparse matrices.

5.2 Eigenvalue Solvers

The discretized Schrödinger equation becomes a matrix eigenvalue problem $H\psi = E\psi$. Sparse eigenvalue solvers from `scipy.sparse.linalg` (specifically `eigsh` for Hermitian matrices) are used to find the lowest energy eigenvalues and corresponding eigenvectors efficiently.

5.3 Linear System Solvers

The discretized Poisson equation becomes a linear system $A\phi = b$. Sparse linear solvers from `scipy.sparse.linalg` (specifically `spsolve` for direct solution or iterative solvers like GMRES (`gmres`) or LGMRES (`lgmres`) as fallbacks) are used to find the potential ϕ. Dirichlet boundary conditions are implemented by modifying the corresponding rows and right-hand side vector of the linear system matrix.

5.4 Spectral Method (Poisson)

The `solve_poisson_2d_spectral` function uses Fast Fourier Transforms (FFTs) to solve the Poisson equation in k-space. The equation $\nabla^2\phi = -\rho/\epsilon$ becomes $-K^2\Phi_k = -P_k/\epsilon$ in Fourier space, where $K^2 = k_x^2 + k_y^2$, and Φ_k, P_k are the Fourier transforms of ϕ, ρ. This method is efficient for uniform grids and periodic boundary conditions. The DC component ($K = 0$) is handled separately by setting $\Phi_k(0,0) = 0$, which corresponds to setting the average potential to zero.

6 Benchmark Results and Numerical Schemes

6.1 Benchmark Summary

The performance of different numerical schemes for solving the Schrödinger-Poisson equations was benchmarked using the `benchmark_solver_schemes.py` script. The script tests different Poisson solvers (Finite Difference, Spectral) and Schrödinger solvers (various configurations of `eigsh` and `lobpcg`) under different scenarios, including cold starts with random voltages and warm starts with perturbed voltages.

Figure **??** shows a summary of the benchmark results across all scenarios. The plot illustrates the average total time, average Schrödinger solver time per iteration, and average Poisson solver time per iteration for each combination of solvers and scenarios. The number of converged samples out of the total number of samples is also indicated on the plot.

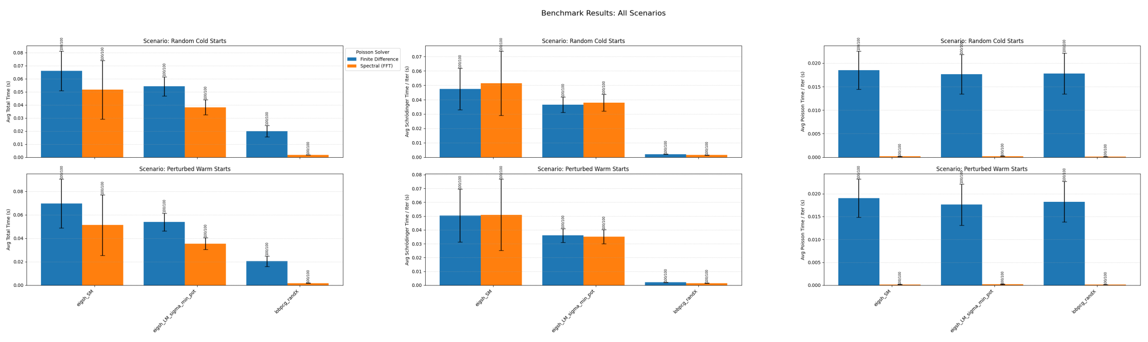

Figure 3: Benchmark results for different Schrödinger-Poisson solver schemes across various scenarios. The plot shows the average total time, average Schrödinger solver time per iteration, and average Poisson solver time per iteration. The number of converged samples is also indicated.

6.2 Numerical Schemes: Background and Implementation Details

The simulation suite employs several numerical schemes to solve the Schrödinger and Poisson equations. This section provides a brief overview of these schemes, including their background and implementation details.

6.2.1 Schrödinger Solvers

Finite Difference Method The Schrödinger equation is discretized using the finite difference method (FDM). The second derivative is approximated using a central difference scheme:

$$\frac{d^2\psi}{dx^2} \approx \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{dx^2}$$

In 2D, the Laplacian is approximated using a 5-point stencil. This discretization transforms the Schrödinger equation into a sparse matrix eigenvalue problem.

`eigsh` (ARPACK) The `eigsh` function from `scipy.sparse.linalg` is used to solve the sparse matrix eigenvalue problem arising from the discretized Schrödinger equation. `eigsh` is a wrapper around the ARPACK library, which implements the Implicitly Restarted Arnoldi Method (IRAM).

ARPACK is designed for finding a few eigenvalues and eigenvectors of large sparse matrices. It is particularly efficient for finding eigenvalues at the extreme ends of the spectrum (i.e., smallest or largest magnitude).

Key parameters for `eigsh` include:

- `k`: The number of eigenvalues and eigenvectors to find.
- `which`: Specifies which eigenvalues to find (e.g., `'SM'` for smallest magnitude, `'LM'` for largest magnitude).
- `sigma`: Specifies a shift value for finding eigenvalues near a particular value. Using `sigma` can improve convergence for interior eigenvalues.
- `tol`: The desired relative accuracy for the eigenvalues.
- `maxiter`: The maximum number of iterations allowed.

`lobpcg` The `lobpcg` function from `scipy.sparse.linalg` implements the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) method. LOBPCG is an iterative method for finding the lowest (or highest) eigenvalues and corresponding eigenvectors of a symmetric (or Hermitian) matrix.

LOBPCG is a block Krylov subspace method, which means it operates on a block of vectors simultaneously. This can lead to faster convergence compared to single-vector methods like the power method or the Lanczos method.

Key parameters for `lobpcg` include:

- `X`: An initial guess for the eigenvectors. The shape of `X` determines the number of eigenvalues and eigenvectors to find.
- `B`: A preconditioner matrix. A good preconditioner can significantly improve convergence.
- `M`: A constraint matrix.
- `tol`: The desired relative accuracy for the eigenvalues.
- `maxiter`: The maximum number of iterations allowed.

6.2.2 Poisson Solvers

Finite Difference Method The Poisson equation is also discretized using the finite difference method. The Laplacian is approximated using a central difference scheme, similar to the Schrödinger equation. Dirichlet boundary conditions (potential fixed at boundaries) are applied by modifying the corresponding rows and right-hand side vector of the linear system matrix.

`spsolve` The `spsolve` function from `scipy.sparse.linalg` is used to solve the sparse linear system arising from the discretized Poisson equation. `spsolve` is a direct solver, which means it computes an exact solution (up to machine precision) in a finite number of steps.

GMRES The `gmres` function from `scipy.sparse.linalg` implements the Generalized Minimal Residual (GMRES) method. GMRES is an iterative method for solving non-symmetric linear systems. It is particularly useful when the matrix is large and sparse, and a direct solver is not feasible.

Spectral Method (FFT) The spectral method solves the Poisson equation in Fourier space using Fast Fourier Transforms (FFTs). The equation $\nabla^2\phi = -\rho/\epsilon$ becomes $-K^2\Phi_k = -P_k/\epsilon$ in Fourier space, where $K^2 = k_x^2 + k_y^2$, and Φ_k, P_k are the Fourier transforms of ϕ, ρ. This method is efficient for uniform grids and periodic boundary conditions. The DC component ($K = 0$) is handled separately by setting $\Phi_k(0,0) = 0$, which corresponds to setting the average potential to zero.

7 Conclusion

This simulation suite provides a flexible framework for modeling quantum dot devices formed by top gates on a 2DEG. By solving the Schrödinger and Poisson equations self-consistently, it can predict electron distributions, energy levels, and characteristic experimental signatures like pinch-off curves and charge stability diagrams under various gate configurations. The inclusion of different Poisson solvers and sweep strategies allows for exploration of numerical techniques and optimization of simulation performance.