Superconducting Circuit Quantization: A Complete Guide

From Classical Circuits to Quantum Hamiltonians

This document covers every piece of physics and mathematics behind your quantum circuit simulation codebase. It's written for someone with a strong math and CS background but limited physics intuition. Each section maps directly to a component of your code.

Part 1: The Lagrangian — Why Circuits Are Mechanics

1.1 The Core Analogy

Every physics problem starts with choosing the right variables and writing down the energy. For circuits, there's a remarkable fact: **a circuit with capacitors and inductors is mathematically identical to a system of masses and springs.**

Mechanics	Circuits
Position x	Node flux Φ
Velocity ẋ	Voltage Φ̇ (= V)
Momentum p	Charge Q
Mass m	Capacitance C
Spring constant k	Inverse inductance 1/L
Kinetic energy ½mẋ²	Capacitive energy ½CΦ̇²
Potential energy ½kx²	Inductive energy ½Φ²/L

This isn't a loose metaphor — the equations are literally identical. This is why your code can borrow the entire framework of Lagrangian and Hamiltonian mechanics from physics.

1.2 What Is Node Flux?

In your code, the fundamental variable is **node flux** Φ. This might seem abstract, but it has a simple definition:

$$\Phi(t) = \int_{-\infty}^{t} V(t')\, dt'$$

Node flux is the time-integral of voltage at a node. Its time derivative is voltage: $\dot{\Phi} = V$. This is your generalized coordinate — analogous to position in mechanics.

Why flux and not voltage? Because the Lagrangian formalism needs a coordinate and its derivative. If we used voltage V directly, we'd need its integral (flux) anyway for the inductor energy. Using flux as the coordinate means voltage comes for free as the derivative.

1.3 What Is Charge in This Framework?

Just as momentum $p = m\dot{x}$ in mechanics, charge is the **conjugate momentum** to flux:

$$Q = \frac{\partial \mathcal{L}}{\partial \dot{\Phi}} = C\dot{\Phi} = CV$$

This is just the capacitor equation $Q = CV$, but now understood as a statement about conjugate variables. This relationship is what makes the mass–capacitance analogy exact.

1.4 The Lagrangian for a Single LC Circuit

For a single node connected to ground through a capacitor C and inductor L:

$$\mathcal{L} = T - U = \frac{1}{2}C\dot{\Phi}^2 - \frac{1}{2}\frac{\Phi^2}{L}$$

The first term is kinetic energy (capacitive energy, stored in the electric field). The second term is potential energy (inductive energy, stored in the magnetic field).

This is your `get_kinetic_energy` and `get_potential_energy` methods. The Lagrangian is their difference, computed by `get_lagrangian`.

1.5 Euler-Lagrange Equations → Equations of Motion

The Euler-Lagrange equation says:

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{\Phi}} = \frac{\partial \mathcal{L}}{\partial \Phi}$$

For our LC circuit:

- Left side: $d/dt\,(C\dot{\Phi}) = C\ddot{\Phi}$
- Right side: $-\Phi/L$

So: $C\ddot{\Phi} = -\Phi/L$, which gives $\ddot{\Phi} = -(1/LC)\Phi$.

This is a simple harmonic oscillator with ω² = 1/LC. The circuit oscillates — charge sloshes back and forth between the capacitor and inductor, just like a mass bouncing on a spring.

1.6 Multi-Node Circuits: Matrices

When you have multiple nodes (a, b, c, ...), each node gets its own flux variable. The Lagrangian generalizes to:

$$\mathcal{L} = \frac{1}{2}\dot{\vec{\Phi}}^T \mathbf{C}\dot{\vec{\Phi}} - \frac{1}{2}\vec{\Phi}^T \mathbf{M}_L^{-1}\vec{\Phi}$$

where **C** is the capacitance matrix and **M_L⁻¹** is the inverse inductance matrix. These are the matrices your `_build_matrices` method constructs.

How the matrices are built (your code):

For each capacitive branch between nodes j and k with capacitance C_branch:

- Off-diagonal: C[j][k] += −C_branch and C[k][j] += −C_branch
- After all branches: C[j][j] = −sum of row j

This structure ensures that the energy only depends on **flux differences** between nodes, not absolute fluxes. If you shift all fluxes by the same constant, the energy doesn't change — this is the circuit analog of translational invariance. The ground node is then removed (row/column 0 deleted) because it's the reference point (Φ_ground = 0), leaving you with a (P−1) × (P−1) matrix.

1.7 Normal Modes

The multi-node equation of motion is:

$$\mathbf{C}\ddot{\vec{\Phi}} = -\mathbf{M}_L^{-1}\vec{\Phi}$$

$$\ddot{\vec{\Phi}} = -\mathbf{C}^{-1}\mathbf{M}_L^{-1}\vec{\Phi}$$

Looking for oscillating solutions $\vec{\Phi}(t) = \vec{v}e^{i\omega t}$ gives the eigenvalue problem:

$$\mathbf{C}^{-1}\mathbf{M}_L^{-1}\vec{v} = \omega^2\vec{v}$$

This is your `_build_omega_squared` and the `np.linalg.eig` call. The eigenvalues are the squared frequencies ω² of the normal modes, and the eigenvectors describe which combinations of node fluxes oscillate together. Each normal mode is an independent oscillator.

For a single LC node, this collapses to the scalar ω² = 1/LC.

Part 2: The Hamiltonian — Preparing for Quantization

2.1 From Lagrangian to Hamiltonian

The Hamiltonian is obtained by a Legendre transform:

$$H = \sum_i Q_i \dot{\Phi}_i - \mathcal{L}$$

Using $Q = C\dot{\Phi}$ (so $\dot{\Phi} = C^{-1}Q$), this becomes:

$$H = \frac{1}{2}\vec{Q}^T \mathbf{C}^{-1} \vec{Q} + \frac{1}{2}\vec{\Phi}^T \mathbf{M}_L^{-1} \vec{\Phi}$$

The first term is kinetic energy expressed in terms of charge (momentum), the second is potential energy in terms of flux (position). **This is your `get_hamiltonian` method.**

2.2 Why Switch to the Hamiltonian?

The Lagrangian uses coordinates and velocities (Φ, $\dot{\Phi}$). The Hamiltonian uses coordinates and momenta (Φ, Q). Quantization requires the Hamiltonian because the fundamental quantum mechanical recipe is:

1. Take the classical Hamiltonian $H(\Phi, Q)$
2. Promote Φ and Q to operators: $\Phi \to \hat{\Phi}$, $Q \to \hat{Q}$
3. Impose the canonical commutation relation: $[\hat{\Phi}, \hat{Q}] = i\hbar$
4. The quantum Hamiltonian $\hat{H}$ determines the energy levels

This is called **canonical quantization**. You can't do it directly from the Lagrangian because it involves velocities, not momenta, and the commutation relation is between coordinate and momentum.

Part 3: The Josephson Junction — Why Qubits Exist

3.1 What Is a Josephson Junction?

A Josephson junction is two superconductors separated by a thin insulating barrier. In a superconductor, all the electrons form Cooper pairs (two electrons bound together) that all share a single quantum mechanical phase. The junction lets Cooper pairs tunnel through the barrier one at a time.

The key physics is captured by two equations:

Josephson current-phase relation: $I = I_c \sin(\phi)$

Josephson voltage-phase relation: $V = (\Phi_0/2\pi)\, d\phi/dt$

where ϕ is the superconducting phase difference across the junction, I_c is the critical current (maximum supercurrent the junction can carry), and $\Phi_0 = \hbar/2e$ is the reduced flux quantum.

3.2 The Josephson Energy

The energy stored in a Josephson junction is:

$$U_{JJ} = -E_J \cos(\varphi)$$

where $E_J = \hbar I_c/2e$ is the Josephson energy. Compare this to an inductor's energy $\frac{1}{2}\Phi^2/L$, which is a parabola. The cosine is periodic — it repeats every 2π. This periodicity reflects the fact that the phase is an angle, defined modulo 2π.

In terms of node flux (using $\phi = \Phi/\Phi_0$):

$$U_{JJ} = -E_J \cos\left(\frac{\Phi}{\Phi_0}\right)$$

This is the non-linear term in your `get_potential_energy` method.

3.3 The Josephson Junction as a Nonlinear Inductor

For small phase differences, $\cos(\phi) \approx 1 - \phi^2/2$, so:

$$U_{JJ} \approx -E_J + \frac{E_J}{2}\varphi^2 = -E_J + \frac{\Phi^2}{2L_J}$$

where $L_J = \Phi_0^2/E_J$ is the Josephson inductance. **This is your `get_LJ` static method.** At small oscillations, the junction acts like a linear inductor with inductance L_J.

But at larger amplitudes, the higher-order terms in the Taylor expansion of cosine matter:

$$\cos(\varphi) = 1 - \frac{\varphi^2}{2} + \frac{\varphi^4}{24} - \cdots$$

The ϕ^4 term and beyond make the potential **anharmonic** — the energy levels are not evenly spaced. This is the entire reason superconducting qubits work, as explained next.

3.4 Why Anharmonicity Is Essential for Qubits

A harmonic oscillator (linear LC circuit) has evenly spaced energy levels: $E_n = \hbar\omega(n + \frac{1}{2})$. The gap between any two adjacent levels is always $\hbar\omega$. This means if you send in a signal at frequency ω to excite the $|0\rangle \to |1\rangle$

transition, you simultaneously excite $|1\rangle \to |2\rangle$, $|2\rangle \to |3\rangle$, and every other transition. You cannot selectively address just the lowest two levels. A harmonic oscillator is useless as a qubit.

The Josephson junction's cosine potential breaks this degeneracy. The energy levels become:

$$E_{01} \neq E_{12} \neq E_{23} \neq \cdots$$

Now you can send a signal at exactly the frequency $f_{01} = (E_1 - E_0)/h$ and it will only excite the $|0\rangle \to |1\rangle$ transition, because $f_{01} \neq f_{12}$. You can treat the two lowest levels as an effective two-level system — a qubit.

The **anharmonicity** $\alpha = E_{12} - E_{01}$ measures how different the qubit transition is from the next one up. For a transmon, α is typically negative and around 200–300 MHz, meaning the $|1\rangle \to |2\rangle$ transition is slightly lower in frequency than $|0\rangle \to |1\rangle$.

3.5 The Junction Also Has a Capacitance

A real Josephson junction isn't just a nonlinear inductor — the two superconducting plates separated by an insulator also form a parallel plate capacitor with capacitance C_J. So a junction is modeled as a nonlinear inductance in parallel with a capacitance.

This is why in your code, `JosephsonJunction` appears in both the capacitive and inductive subgraphs. Your `_build_sub_graphs` method includes JJ in both:

```python
if type(branch) in [Capacitor, JosephsonJunction]:
    capacitive_branches.append(branch)
if type(branch) in [Inductor, JosephsonJunction]:
    inductive_branches.append(branch)
```

Part 4: The Graph Representation — Encoding Circuit Topology

4.1 Why a Graph?

A circuit is naturally a graph: nodes are vertices, components are edges. Your `graph_rep` dictionary encodes this:

```python
```

```
graph_rep = {
    'nodes': ['a', 'b'],          # non-ground nodes
    'capacitors': [('a','b',C)],   # capacitive edges
    'inductors': [('a','gnd',L)],  # inductive edges
    'josephson_junctions': [('a','gnd',EJ,CJ)],  # JJ edges
    'external_flux': {}            # flux threading loops
}
```

Your `_build_master_graph` method converts this into `Node` and `Branch` objects with bidirectional references (each node knows its branches, each branch knows its nodes). This lets you traverse the circuit topology.

4.2 Symmetry Breaking

There's a subtlety: the capacitance matrix must be invertible (you need C^{-1} for the Hamiltonian). If a node has no capacitive connection to ground (directly or indirectly), the matrix can be singular.

Your code handles this by adding a tiny parasitic capacitor (1e-20 F) from every node to ground if it doesn't already have a capacitive branch. This is physically reasonable — real circuits always have stray capacitance — and it's numerically necessary to make C invertible.

4.3 Active vs. Passive Nodes

A node is **active** if it has at least one inductive branch (meaning its flux appears in the potential energy and has a restoring force). A node is **passive** if it only has capacitive connections (its flux doesn't appear in the potential energy).

Active nodes are the "real" quantum degrees of freedom. Passive nodes can in principle be eliminated through the equations of motion — their flux is determined by the active nodes. Your `self.N` counts active nodes, representing the true number of degrees of freedom for the quantum system.

Part 5: Quantization in the Charge Basis

5.1 The Single-Node Transmon Hamiltonian

For a single node connected to ground through a Josephson junction (with capacitance C_J) and possibly a shunt capacitor C_S, the total capacitance is C = C_J + C_S. The Hamiltonian is:

$$H = \frac{Q^2}{2C} - E_J \cos\left(\frac{\Phi}{\Phi_0}\right)$$

Since charge comes in units of Cooper pairs (Q = 2en, where n is an integer), we define the charging energy:

$$E_C = \frac{e^2}{2C}$$

and the Hamiltonian becomes:

$$H = 4E_C\hat{n}^2 - E_J\cos(\hat{\varphi})$$

where n̂ is the number of Cooper pairs on the island and $\hat{\varphi} = \hat{\Phi}/\Phi_0$ is the dimensionless phase.

5.2 The Charge Basis

We choose to work in the eigenstates of n̂, denoted |n⟩ where n = ...,-2,-1,0,1,2,... Each state |n⟩ represents exactly n Cooper pairs on the superconducting island.

In this basis:

- **n̂ is diagonal:** n̂|n⟩ = n|n⟩, so the matrix representation is diag(...,–2,–1,0,1,2,...).
- **The cosine becomes off-diagonal.** This requires understanding the phase operator.

5.3 The Phase Operator and the Cosine

The key relationship is $[\hat{\varphi}, \hat{n}] = i$ (this follows from $[\hat{\Phi}, \hat{Q}] = i\hbar$ and the definitions $\varphi = \Phi/\Phi_0$, Q = 2en). This commutation relation tells us that $\hat{\varphi}$ is the "momentum" conjugate to n, or equivalently, n̂ is the "momentum" conjugate to φ.

The exponential e^{i$\hat{\varphi}$} is a **shift operator** on the charge basis:

$$e^{i\hat{\varphi}}|n\rangle = |n+1\rangle$$

This can be proven from the commutation relation (it's the same math as e^{i$\hat{p}x_0/\hbar$} shifting position eigenstates). Physically, advancing the phase by one unit corresponds to one Cooper pair tunneling across the junction.

Now we can evaluate the cosine:

$$\cos(\hat{\varphi}) = \frac{e^{i\hat{\varphi}} + e^{-i\hat{\varphi}}}{2}$$

In the charge basis:

$$\cos(\hat{\varphi})|n\rangle = \frac{|n+1\rangle + |n-1\rangle}{2}$$

So the cosine operator connects each charge state to its neighbors. Its matrix representation is ½ on the super- and sub-diagonals, with zeros everywhere else.

5.4 The Full Hamiltonian Matrix

Putting it together, in the charge basis truncated to n ∈ {−n_cut, ..., n_cut}:

$$\hat{H} = 4E_C\hat{n}^2 - \frac{E_J}{2}\,(\text{superdiagonal of 1s} + \text{subdiagonal of 1s})$$

This is exactly what your `_quantize` method builds:

```
python

H = (4 * EC * (n_hat @ n_hat)) - 0.5 * EJ * (np.diag(np.ones(dim-1), 1) + np.diag(np.ones(dim-1), -1))
```

The diagonal part $4E_C n^2$ assigns increasing energy to states with more Cooper pairs (it costs energy to pile up charge). The off-diagonal $-E_J/2$ terms allow tunneling between adjacent charge states (Cooper pairs hopping across the junction). Competition between these two terms determines the character of the eigenstates.

5.5 The Transmon Regime

The ratio E_J/E_C determines the physics:

- **$E_J/E_C \ll 1$ (charge qubit regime):** The charging energy dominates. Eigenstates are close to pure charge states $|n\rangle$. Very sensitive to charge noise (bad for quantum computing).
- **$E_J/E_C \gg 1$ (transmon regime):** The Josephson tunneling dominates. Eigenstates are broad superpositions spread over many charge states. Insensitive to charge noise (good for quantum computing).

The transmon typically operates at $E_J/E_C \approx 50$–100. The tradeoff is that higher E_J/E_C reduces anharmonicity (levels become more evenly spaced), making it harder to address just the qubit transition. But the exponential suppression of charge noise sensitivity is worth the modest reduction in anharmonicity.

5.6 Diagonalization

Your `_diagonalize` method calls `np.linalg.eigh(H)` to find the eigenvalues (energy levels) and eigenvectors (stationary states). The eigenvalues $E_0, E_1, E_2, \ldots$ are the allowed energies, and the eigenvectors express these states as superpositions in the charge basis.

The qubit frequency is $f_{01} = (E_1 - E_0)/h$, typically 4–8 GHz for a transmon.

5.7 Change of Basis

Your `_change_basis` method transforms the charge operator $\hat{n}$ from the charge basis to the energy eigenbasis:

$$\hat{n}_{\text{energy}} = S^\dagger \hat{n} S$$

where S is the matrix of eigenvectors (columns of `self.states`). This is needed for the time evolution, where you work in the energy basis but the drive couples to charge.

Part 6: Unified Quantization — The Flux Operator in the Charge Basis

6.1 The Problem

Your original `_quantize` only handles the transmon (JJ, no linear inductor). For a linear LC circuit, you need the Φ^2 term. For a fluxonium (JJ + inductor), you need both. A unified approach requires the flux operator $\hat{\Phi}$ in the charge basis.

6.2 The Flux Operator

From $[\hat{\phi}, \hat{n}] = i$, we know $e^{i\hat{\phi}}$ is the shift operator. The phase operator itself can be approximated via:

$$\hat{\varphi} = \frac{1}{2i}\left(e^{i\hat{\varphi}} - e^{-i\hat{\varphi}}\right)$$

This is a finite-difference approximation ($\sin(\hat{\phi})$ actually, treating ϕ as if it were on a lattice of integers). In the charge basis, this is an antisymmetric tridiagonal matrix with $\pm 1/2i$ on the off-diagonals.

Since $\hat{\Phi} = \Phi_0\,\hat{\phi}$, the flux operator is:

$$\hat{\Phi}_{nm} = \frac{\Phi_0}{2i}(\delta_{n,m-1} - \delta_{n,m+1})$$

Then the inductive potential $\frac{1}{2}\Phi^2/L$ requires $\hat{\Phi}^2$, which is just this matrix squared.

6.3 The Unified Hamiltonian

For any single-node circuit:

$$\hat{H} = 4E_C\hat{n}^2 + \frac{1}{2L}\hat{\Phi}^2 - E_J\cos(\hat{\varphi})$$

where you include whichever terms are present (the L term for inductors, the cosine for JJs, or both for fluxonium). All three terms have known matrix representations in the charge basis, so you build them all in the same framework and diagonalize once.

6.4 Convergence Caveat

The finite-difference approximation of $\hat{\phi}$ introduces truncation error. For the transmon (no Φ^2 term), only the exponentials $e^{\pm i\hat{\phi}}$ appear, which are exact shift operators regardless of truncation — you just need n_cut

large enough to capture the wavefunction tails. For the LC circuit, the Φ^2 term uses the approximate $\hat{\phi}$, so you need a larger n_cut for the eigenvalues to converge to the analytic $\hbar\omega(n + ½)$. Always check convergence by increasing n_cut and watching eigenvalues stabilize.

Part 7: External Flux

7.1 Flux Through a Loop

If your circuit has a loop (e.g., two inductors connecting the same pair of nodes via different paths), you can thread an external magnetic flux Φ_ext through that loop. This modifies the potential energy because the relevant variable for each inductor becomes the **gauge-invariant phase difference**, not just the node flux difference.

In your code, this appears in `get_potential_energy` as the `offset_dict` terms:

```python
for (node1_label, node2_label), offset in self.offset_dict.items():
    ...
    term_2 += ((node_flux[j] - node_flux[k]) * offset) / inductance
```

This adds a linear term to the potential energy, shifting the equilibrium point of the parabola. It's how you bias a SQUID or tune a flux-tunable transmon.

Part 8: Time Evolution — Driving the Qubit

8.1 The Physical Setup

You have a transmon sitting at its ground state $|0\rangle$. You want to rotate it to $|1\rangle$ (this is a quantum NOT gate, or X gate). To do this, you apply a time-dependent drive — a signal at (or near) the qubit's transition frequency.

In the SFQ approach, this drive consists of a train of very short voltage pulses (Single Flux Quantum pulses, each carrying exactly one flux quantum $\Phi_0 = h/2e$ of magnetic flux). Each pulse gives the qubit a small kick, and if the pulses arrive at the right frequency, the kicks accumulate coherently.

8.2 The Driven Hamiltonian

The total Hamiltonian during the drive is:

$$\hat{H}(t) = \hat{H}_0 + A(t)\hat{n}$$

where $\hat{H}_0$ is the time-independent qubit Hamiltonian (with eigenvalues $E_0, E_1, E_2, ...$), A(t) is the time-dependent pulse envelope, and $\hat{n}$ is the charge operator (the thing the voltage drive couples to).

Why charge? A voltage pulse V(t) delivers energy V(t)·Q to the circuit. Since Q = 2e$\hat{n}$, the drive term is proportional to V(t)$\hat{n}$. Your A(t) absorbs the proportionality constants.

8.3 Working in the Energy Eigenbasis

Your `crank_nicolson` method works in the energy eigenbasis, where $\hat{H}_0$ is diagonal. The advantages are:

1. **$\hat{H}_0$ is just `np.diag(energies)`** — trivial to construct.
2. **Truncation is clean** — you keep the lowest `dim_sub` levels (typically 3–5), knowing these are the physically relevant states. In the charge basis, you'd need the full 2n_cut + 1 dimensions because low-energy eigenstates span many charge states.
3. **Physical interpretation is direct** — the state vector components immediately give you the probability of being in $|0\rangle$, $|1\rangle$, $|2\rangle$, etc.

The charge operator in this basis is `n_hat_energy = S†·n_hat·S`, precomputed by `_change_basis`.

8.4 The SFQ Pulse Train

Your pulse envelope is a sum of Gaussians:

$$A(t) = A_0 \sum_{k=0}^{N_{\text{pulses}}-1} \exp\left(-\frac{(t - kT_{\text{drive}})^2}{2\sigma^2}\right)$$

Each Gaussian represents one SFQ pulse. The parameters are:

- **T_drive = 1/f_drive:** spacing between pulses. For resonant driving, f_drive should match the qubit frequency $f_{01} = (E_1 - E_0)/h$.
- **σ:** pulse width. Real SFQ pulses are extremely narrow (a few picoseconds).
- **$A_0 = 0.01 \times (E_1 - E_0)$:** pulse amplitude. Set small so each pulse is a gentle perturbation. This ensures the drive stays within the perturbative regime where only the qubit levels $|0\rangle$ and $|1\rangle$ participate significantly, and leakage to $|2\rangle$ is minimal.
- **N_pulses:** total number of pulses. More pulses = longer drive = larger total rotation angle.

8.5 The Crank-Nicolson Method

Time evolution is governed by the Schrödinger equation:

$$i\hbar \frac{\partial}{\partial t}|\psi\rangle = \hat{H}(t)|\psi\rangle$$

The formal solution over one timestep dt is:

$$|\psi(t+dt)\rangle = e^{-i\hat{H}dt/\hbar}|\psi(t)\rangle$$

Computing the matrix exponential exactly at every step is expensive. Instead, you approximate it. The simplest approach — forward Euler, replacing the exponential with I – iĤdt/$\hbar$ — is cheap but **not unitary**. It doesn't preserve the norm of the state vector, so probabilities stop summing to 1 and the simulation becomes unphysical.

The Crank-Nicolson method fixes this by evaluating the Hamiltonian at the midpoint of the timestep:

$$\left(I + \frac{i\hat{H}_{\text{mid}}dt}{2\hbar}\right)|\psi(t+dt)\rangle = \left(I - \frac{i\hat{H}_{\text{mid}}dt}{2\hbar}\right)|\psi(t)\rangle$$

This is **unconditionally unitary** when H is Hermitian — the evolution matrix $A^{-1}B$ is unitary regardless of step size. The cost is solving a linear system at each step (`np.linalg.solve`), but for small matrices (dim_sub ≈ 3–5), this is negligible.

In your code:

```
python

A = I + (((1j*dt) / (2*hbar)) * H_mid)
B = I - (((1j*dt) / (2*hbar)) * H_mid)
psi = np.linalg.solve(A, B @ psi)
```

Ĥ_mid is the total Hamiltonian evaluated at t + dt/2, using the pulse amplitude at that midpoint. This midpoint evaluation is important because A(t) varies during the timestep.

8.6 What Rabi Oscillations Are

When you drive a qubit resonantly (f_drive = f_{01}), the population oscillates sinusoidally between |0⟩ and |1⟩. These are **Rabi oscillations**. The rate of oscillation (Rabi frequency) is proportional to the drive amplitude.

Your code tracks P_0, P_1, P_2 at each timestep — the probabilities of finding the qubit in |0⟩, |1⟩, and |2⟩ respectively. For a good qubit:

- P_0 and P_1 should oscillate cleanly between 0 and 1
- P_2 (leakage to the second excited state) should remain near zero

If P_2 grows significantly, the drive is too strong, the frequency is wrong, or the anharmonicity is insufficient.

8.7 Why Driving Frequency Matters

The drive at frequency f_drive creates transitions at that frequency. If f_drive matches $E_1 - E_0$, the $|0\rangle \rightarrow |1\rangle$ transition is resonant and absorption is efficient. If f_drive matches $E_2 - E_1$ instead, you'd drive the $|1\rangle \rightarrow |2\rangle$ transition (leakage). If f_drive matches neither, the kicks from successive pulses interfere destructively and nothing happens — the qubit barely responds.

The anharmonicity $\alpha = f_{12} - f_{01}$ provides the "spectral selectivity" that lets you address just the qubit transition. The larger $|\alpha|$, the easier it is to avoid leakage.

Part 9: Gate Fidelity — The Research Frontier

9.1 What Is Gate Fidelity?

A quantum gate is a unitary operation on the qubit. An X gate (NOT gate) takes $|0\rangle \rightarrow |1\rangle$ and $|1\rangle \rightarrow |0\rangle$. After your pulse sequence, you've applied some actual unitary U_actual to the qubit. The **gate fidelity** measures how close this is to the target unitary U_target:

$$F = \frac{1}{d^2} \left| \mathrm{Tr}(U_{\text{target}}^{\dagger} U_{\text{actual}}) \right|^2$$

where d = 2 for a qubit. F = 1 means perfect gate, F < 1 means errors.

9.2 Sources of Infidelity in SFQ Control

Several things can reduce fidelity:

- **Off-resonance:** If f_drive doesn't exactly match f_{01}, the rotation axis is tilted and you don't get a clean X gate.
- **Leakage:** Population escaping to $|2\rangle$ and beyond, caused by the drive being too strong or too spectrally broad.
- **Pulse number error:** If you don't apply exactly the right number of pulses, the rotation angle isn't exactly π.
- **Pulse shape effects:** Real SFQ pulses have finite width, affecting the spectral content.

9.3 The Optimization Problem

The research goal is: find the pulse parameters (f_drive, N_pulses, individual amplitudes, timing) that maximize gate fidelity. This requires:

1. **Computing U_actual** — evolve each basis state through the pulse sequence to build the full propagator matrix.
2. **Computing fidelity** — compare U_actual to U_target.

3. **Optimizing** — search over parameters to maximize F.

Established approaches include sweeping pulse parameters with grid search or Nelder-Mead, and more sophisticated methods like **GRAPE** (Gradient Ascent Pulse Engineering), which computes gradients of the fidelity with respect to pulse amplitudes at each timestep and does gradient ascent.

Part 10: Physical Constants and Units

10.1 Constants in Your Code

Symbol	Name	Value	Role
e	Elementary charge	1.602×10^{-19} C	Charge of one electron
2e	Cooper pair charge	3.204×10^{-19} C	Charge carrier in superconductors
$\hbar$	Reduced Planck constant	1.055×10^{-34} J·s	Quantum of action
h	Planck constant	6.626×10^{-34} J·s	$h = 2\pi\hbar$
$\Phi_0 = \hbar/2e$	Reduced flux quantum	3.291×10^{-16} Wb	Natural flux unit for Cooper pairs
$E_C = e^2/2C$	Charging energy	~ 200 MHz (varies)	Energy cost of one extra electron
E_J	Josephson energy	~ 10–50 GHz (varies)	Tunneling energy scale

10.2 Why Energy Is Often Expressed in Frequency

In superconducting circuits, energies are almost always quoted in frequency units (GHz), using $E = hf$. This is because the energy scales correspond directly to microwave frequencies:

- A qubit at 5 GHz has transition energy $E_{01} = h \times 5\ \text{GHz} \approx 3.3 \times 10^{-24}$ J
- The charging energy $E_C \approx 200$ MHz corresponds to the energy to add one extra electron
- The anharmonicity $\alpha \approx -300$ MHz is the frequency difference between transitions

Your code works in SI (Joules), which is why you see conversions like `/ e / 1e-3` (converting Joules to millielectronvolts) in the plotting functions.

Summary: The Full Pipeline

Here is how every component connects, from circuit definition to quantum dynamics:

1. **Define circuit topology** (`graph_rep` dictionary) → Graph of nodes and branches
2. **Build matrices** (`_build_matrices`) → Capacitance matrix **C** (mass) and inverse inductance matrix $\mathbf{M\_L^{-1}}$ (spring constants) from the graph topology, encoding Kirchhoff's laws
3. **Compute normal modes** (`_build_omega_squared`) → $\omega^2 = \mathbf{C}^{-1}\mathbf{M}\_L^{-1}$ gives the classical oscillation frequencies via eigenvalue decomposition
4. **Write the Hamiltonian** (`get_hamiltonian`) → $H = \frac{1}{2}Q^T \mathbf{C}^{-1} Q + \frac{1}{2}\Phi^T \mathbf{M}\_L^{-1} \Phi - \Sigma E\_J \cos(\Phi/\Phi_0)$, the classical energy function in terms of charge and flux
5. **Quantize** (`_quantize`) → Promote flux and charge to operators in the charge basis. Build the Hamiltonian matrix: diagonal $4E\_C\, n^2$ (charging) + off-diagonal tunneling terms (JJ cosine and/or inductor Φ^2)
6. **Diagonalize** (`_diagonalize`) → Find energy eigenvalues (spectrum) and eigenstates. The gaps between eigenvalues determine qubit frequency and anharmonicity
7. **Change basis** (`_change_basis`) → Express the charge operator in the energy eigenbasis, needed for coupling to the drive
8. **Time evolution** (`crank_nicolson`) → Apply an SFQ pulse train to the qubit in the energy eigenbasis using unitarity-preserving Crank-Nicolson integration. Track state probabilities over time to observe Rabi oscillations
9. **Assess performance** → Measure gate fidelity, leakage, and optimize pulse parameters. This is the current research frontier: finding the pulse sequence that performs the best quantum gate.