Quantum Control Theory and Implementation

QubitPulseOpt: A Simulation Framework for
Optimal Quantum Control of Two-Level Systems

Rylan Malarchick
Quantum Controls Simulation Project

October 25, 2025

Abstract

This document provides a comprehensive theoretical foundation and implementation guide for the QubitPulseOpt quantum control simulation framework. We develop the mathematical formalism for controlling two-level quantum systems (qubits) using time-dependent electromagnetic pulses, building from first principles to advanced optimal control techniques. The treatment is organized by project phases, with each phase introducing new theoretical concepts alongside their computational implementations. Phase 1 establishes the computational environment, drift Hamiltonian formalism, and control Hamiltonians. Phase 2 implements optimal control algorithms (GRAPE, Krotov) with comprehensive robustness testing. Phase 3 develops advanced pulse shaping techniques, gate compilation, and benchmarking protocols. This document serves as the authoritative reference for all physics, mathematics, and implementation decisions in the QubitPulseOpt framework.

Contents

1 Introduction **4**
1.1 Motivation and Scope . . . 4
1.2 Mathematical Conventions . . . 4

2 Phase 1.1: Computational Infrastructure and Reproducibility **4**
2.1 Overview . . . 4
2.2 Environment Design Philosophy . . . 5
2.2.1 Reproducibility Requirements . . . 5
2.2.2 Virtual Environment vs. Conda . . . 5
2.3 Core Dependencies . . . 5
2.3.1 QuTiP Installation Notes . . . 5
2.4 Repository Structure . . . 6
2.5 Validation Protocol . . . 7
2.6 Git Workflow and Version Control . . . 7
2.7 Summary of Phase 1.1 Deliverables . . . 8

3 Phase 1.2: Drift Hamiltonian and Free Evolution **8**
3.1 Overview . . . 8
3.2 Two-Level Systems and the Qubit Hilbert Space . . . 8
3.3 Physical Derivation of the Drift Hamiltonian . . . 8
3.4 Spectral Properties of the Drift Hamiltonian . . . 9
3.5 Analytical Solution for Time Evolution . . . 9
3.5.1 Time Evolution Operator . . . 9

3.5.2 Evolution of Basis States . . . 10
3.6 Bloch Sphere Representation . . . 10
3.6.1 Bloch Vector Mapping . . . 10
3.6.2 Drift Dynamics on the Bloch Sphere . . . 10
3.7 Numerical Implementation . . . 11
3.7.1 DriftHamiltonian Class . . . 11
3.7.2 TimeEvolution Engine . . . 11
3.8 Validation and Testing . . . 13
3.8.1 Unit Test Coverage . . . 13
3.8.2 Analytical vs. Numerical Comparison . . . 13
3.9 Demonstration Notebook . . . 13
3.10 Summary of Phase 1.2 Deliverables . . . 14

4 Phase 1.3: Control Hamiltonian and Pulse Shaping 14
4.1 Overview . . . 14
4.2 Control Hamiltonian in the Lab Frame . . . 14
4.3 Rotating Frame and Rotating-Wave Approximation . . . 15
4.4 Rabi Oscillations . . . 15
4.4.1 Constant Driving . . . 15
4.4.2 Quantum Gate Synthesis . . . 15
4.5 Pulse Shapes . . . 16
4.5.1 Gaussian Pulse . . . 16
4.5.2 DRAG Pulses . . . 16
4.6 Detuning Effects . . . 16
4.7 Implementation . . . 16
4.7.1 ControlHamiltonian Class . . . 16
4.7.2 Pulse Shape Generators . . . 17
4.8 Validation and Testing . . . 17
4.8.1 Test Coverage . . . 17
4.8.2 Rabi Oscillation Validation . . . 18
4.9 Demonstration Notebook . . . 18
4.10 Summary of Phase 1.3 Deliverables . . . 18

5 Phase 2: Optimal Control Theory 19
5.1 Overview . . . 19
5.2 Optimal Control Problem Formulation . . . 19
5.2.1 Fidelity Metrics . . . 19
5.2.2 Constrained Optimization . . . 19
5.3 GRAPE Algorithm . . . 19
5.3.1 Gradient Computation . . . 19
5.3.2 Update Rule . . . 19
5.4 Krotov's Method . . . 20
5.5 Robustness Analysis . . . 20
5.5.1 Parameter Sensitivity . . . 20
5.5.2 Noise Robustness . . . 20
5.6 Implementation . . . 20

6 Phase 3: Advanced Pulse Shaping and Benchmarking 21
6.1 Overview . . . 21
6.2 Task 1: Advanced Pulse Shaping . . . 21
6.2.1 DRAG Pulses . . . 21
6.2.2 Composite Pulses . . . 21

6.2.3 Adiabatic Techniques . . . 21
6.3 Task 2: Gate Library and Compilation . . . 22
6.3.1 Universal Single-Qubit Gates . . . 22
6.3.2 Euler Decomposition . . . 22
6.3.3 Gate Compilation . . . 22
6.4 Task 3: Enhanced Robustness and Benchmarking . . . 22
6.4.1 Filter Functions . . . 22
6.4.2 Randomized Benchmarking . . . 23
6.4.3 Fisher Information . . . 23
6.5 Implementation Summary . . . 23

7 Future Work **24**
7.1 Planned Extensions . . . 24

8 Conclusion **24**

References **25**

A Appendix: QuTiP API Reference **25**

1 Introduction

1.1 Motivation and Scope

Quantum control theory addresses the fundamental challenge of manipulating quantum systems to achieve desired target states or operations with high fidelity. For two-level systems (qubits)—the basic unit of quantum information—this control is typically achieved through resonant or near-resonant electromagnetic pulses. The QubitPulseOpt framework provides a rigorous simulation environment for:

(i) Modeling qubit dynamics under drift and control Hamiltonians

(ii) Designing and optimizing control pulse shapes

(iii) Analyzing decoherence and relaxation effects

(iv) Implementing gradient-based optimal control algorithms (GRAPE, Krotov)

(v) Validating control fidelity via numerical and analytical methods

This document is structured to mirror the phased development of the QubitPulseOpt codebase, with each section corresponding to a specific development phase. All theoretical results are accompanied by explicit mappings to implementation files, test suites, and demonstration notebooks.

1.2 Mathematical Conventions

Throughout this document, we adopt the following conventions:

- We work in units where $\hbar = 1$ unless explicitly stated otherwise.
- Quantum states are denoted by ket vectors $|\psi\rangle \in \mathcal{H}$, where $\mathcal{H}$ is the Hilbert space (typically $\mathcal{H} = \mathbb{C}^2$ for a qubit).
- Observables and operators are represented by calligraphic or bold letters (e.g., $\hat{H}$, $\hat{\rho}$).
- Time-dependent quantities are explicitly written as functions of t.
- The Pauli matrices are defined as:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{1}$$

- The identity operator on $\mathbb{C}^2$ is denoted $\mathbb{I}$.

2 Phase 1.1: Computational Infrastructure and Reproducibility

2.1 Overview

Before developing quantum control algorithms, we must establish a robust, reproducible computational environment. Phase 1.1 addresses the foundational infrastructure requirements: version control, dependency management, environment isolation, and validation protocols. This phase ensures that all subsequent theoretical developments can be implemented, tested, and reproduced by independent researchers or on different computational platforms.

2.2 Environment Design Philosophy

2.2.1 Reproducibility Requirements

Scientific computing demands bitwise-reproducible results across different machines and time periods. For the QubitPulseOpt framework, reproducibility is achieved through:

1. **Environment Isolation:** All Python dependencies are installed in an isolated virtual environment, preventing version conflicts with system packages or other projects.
2. **Explicit Dependency Specification:** Exact versions of all packages are documented in `environment.yml` (for Conda) and can be captured via `pip freeze` for venv users.
3. **Version Control:** All source code, documentation, and configuration files are tracked via Git and hosted on GitHub (https://github.com/rylanmalarchick/QubitPulseOpt).
4. **Automated Validation:** Setup validation scripts ensure that the environment is correctly configured before scientific work begins.

2.2.2 Virtual Environment vs. Conda

Two primary approaches exist for Python environment management:

`venv` **(Python's built-in virtual environment):** Lightweight, no external dependencies, straightforward activation. Used as the primary environment for QubitPulseOpt.

- **Pros:** Ships with Python 3.3+, minimal overhead, integrates seamlessly with pip.
- **Cons:** Does not manage non-Python dependencies (e.g., BLAS, LAPACK for NumPy/SciPy optimization).

`conda` **(Anaconda/Miniconda):** Cross-language package manager, handles compiled dependencies.

- **Pros:** Manages both Python and system-level libraries (e.g., MKL-optimized NumPy), good for complex scientific stacks.
- **Cons:** Larger installation footprint, potential conflicts between conda and pip packages.

Decision: QubitPulseOpt uses `venv` as the default environment, with `environment.yml` provided as an alternative for Conda users. This decision balances simplicity (venv is universally available) with flexibility (Conda users can replicate the environment exactly).

2.3 Core Dependencies

The QubitPulseOpt framework relies on the following Python packages:

2.3.1 QuTiP Installation Notes

QuTiP (Quantum Toolbox in Python) is the central dependency. Version 5.x introduces significant API changes compared to 4.x:

- **Solver Interface:** The `sesolve` function (Schrödinger equation solver) now returns a `Result` object with state trajectories accessed via `result.states`.
- **Bloch Sphere Plotting:** The `Bloch.add_points` method requires the `meth` parameter to be one of `{'s', 'l', 'm'}` (single point, line, multi-point). Colors are set via `point_color`, `point_marker`, etc., rather than passed directly to `add_points`.

Package	Version	Purpose
QuTiP	5.2.1	Quantum Toolbox in Python; provides quantum state/operator representations, Hamiltonian evolution solvers (Schrödinger, Lindblad), and Bloch sphere visualization.
NumPy	2.3.4	Numerical array operations, linear algebra, Fourier transforms.
SciPy	1.16.2	Advanced scientific computing: ODE solvers, optimization routines, special functions.
Matplotlib	3.10.7	Plotting and visualization of quantum dynamics, control pulses, and convergence metrics.
Jupyter	(latest)	Interactive notebook environment for demonstrations and exploratory analysis.
pytest	(latest)	Unit testing framework; ensures correctness of all modules.
pytest-cov	(latest)	Code coverage analysis for test suites.

Table 1: Core dependencies for QubitPulseOpt with version information and usage descriptions.

- **Quantum Objects:** States and operators are represented as `Qobj` instances. Hermiticity, normalization, and dimensionality are automatically validated.

These API details are critical for Phase 1.2 implementations (drift Hamiltonian evolution) and later phases (control pulse simulations).

2.4 Repository Structure

The QubitPulseOpt repository follows a modular, hierarchical structure:

```
quantumControls/
  src/
    __init__.py
    hamiltonian/
      __init__.py
      drift.py          # Drift Hamiltonian (Phase 1.2)
      evolution.py      # Time evolution engine (Phase 1.2)
      control.py        # Control Hamiltonian (Phase 1.3, future)
    pulses/             # Pulse shapes (future)
    optimization/       # GRAPE, Krotov (future)
    visualization/      # Plotting utilities (future)
  tests/
    unit/
      test_drift.py     # Unit tests for drift.py
      ...
    integration/        # End-to-end tests (future)
  notebooks/
    01_drift_dynamics.ipynb    # Phase 1.2 demonstration
    ...
```

```
scripts/
  validate_setup.sh     # Environment validation
  activate_env.sh       # Activate venv helper
  verify_drift_evolution.py  # Standalone drift test
docs/
  science/
    quantum_control_theory.tex  # This document
  SETUP_COMPLETE.md
  REVIEW_SUMMARY.md
environment.yml       # Conda environment spec
README.md
.gitignore
```

This structure separates concerns: `src/` contains all production code, `tests/` validates correctness, `notebooks/` provides interactive demonstrations, and `docs/` maintains theoretical documentation.

2.5 Validation Protocol

To ensure the environment is correctly configured, we implement a multi-stage validation protocol:

Stage 1: Python Version Check: Verify Python ≥ 3.8 (required for QuTiP 5.x and modern NumPy).

Stage 2: Package Import Test: Attempt to import all core packages (QuTiP, NumPy, SciPy, Matplotlib) and verify versions.

Stage 3: QuTiP Functionality Test: Create a simple quantum state $|0\rangle$, apply a Pauli-X gate, and verify $\sigma_x |0\rangle = |1\rangle$ to within numerical precision.

Stage 4: Numerical Precision Test: Confirm that NumPy/SciPy linear algebra operations achieve machine precision ($\sim 10^{-16}$ for double-precision floats).

The validation script `scripts/validate_setup.sh` orchestrates these checks. A successful run outputs:

```
[PASS] Python version: 3.12.3
[PASS] QuTiP 5.2.1 imported successfully
[PASS] NumPy 2.3.4 imported successfully
[PASS] SciPy 1.16.2 imported successfully
[PASS] Matplotlib 3.10.7 imported successfully
[PASS] Basic QuTiP test: |0> -> |1> via Pauli-X
[PASS] Numerical precision: ||I - I|| = 0.0
All validation checks passed.
```

2.6 Git Workflow and Version Control

The project uses Git for version control with the following practices:

- **Branch Strategy:** Development occurs on the `main` branch initially; feature branches will be used for major new components (e.g., `feature/grape-optimizer`).
- **Commit Messages:** Follow conventional commit format: `type(scope): description`, e.g., `feat(drift): implement analytical propagator for H0`.

- **Remote Repository:** Hosted at https://github.com/rylanmalarchick/QubitPulseOpt. All development is pushed regularly to ensure cloud backup and collaboration readiness.

2.7 Summary of Phase 1.1 Deliverables

- Git repository initialized and pushed to GitHub.
- Virtual environment created with all core dependencies installed.
- Repository structure established (src/, tests/, notebooks/, docs/).
- Validation scripts implemented and passing.
- Documentation of environment setup in README.md and SETUP_COMPLETE.md.

Code Mapping: Configuration files (`environment.yml`, `.gitignore`), validation scripts (`scripts/validate_setup.sh`, `scripts/test_env_simple.py`), and documentation (`docs/SETUP_COMPLETE.m`

3 Phase 1.2: Drift Hamiltonian and Free Evolution

3.1 Overview

Phase 1.2 establishes the theoretical and computational framework for the *drift Hamiltonian* $\hat{H}_0$, which governs the natural evolution of a qubit in the absence of external control fields. Understanding drift dynamics is prerequisite to designing control pulses, as optimal control effectively steers the system away from its natural trajectory toward a desired target state.

We develop the drift Hamiltonian for a qubit in a static magnetic field, derive analytical solutions for time evolution, implement numerical solvers for validation, and demonstrate the equivalence of analytical and numerical methods to machine precision.

3.2 Two-Level Systems and the Qubit Hilbert Space

Definition 3.1 (Qubit)**.** A *qubit* is a two-level quantum system whose state space is the two-dimensional complex Hilbert space $\mathcal{H} = \mathbb{C}^2$. The computational basis states are:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2}$$

Any pure state $|\psi\rangle \in \mathcal{H}$ can be written as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \quad \alpha, \beta \in \mathbb{C}, \quad |\alpha|^2 + |\beta|^2 = 1 \tag{3}$$

The Pauli matrices $\{\sigma_x, \sigma_y, \sigma_z\}$ form a basis for traceless Hermitian operators on $\mathcal{H}$. Together with the identity $\mathbb{I}$, they span the space of all 2×2 Hermitian matrices. Any Hermitian operator (including Hamiltonians) can be decomposed as:

$$\hat{H} = c_0\mathbb{I} + c_x\sigma_x + c_y\sigma_y + c_z\sigma_z, \quad c_i \in \mathbb{R} \tag{4}$$

3.3 Physical Derivation of the Drift Hamiltonian

Consider a qubit (e.g., a spin-1/2 particle or a two-level atom) placed in a static magnetic field $\vec{B} = B_0\hat{z}$ aligned along the z-axis. The interaction Hamiltonian between the qubit's magnetic moment $\vec{\mu}$ and the field is:

$$\hat{H}_0 = -\vec{\mu} \cdot \vec{B} \tag{5}$$

For a spin-1/2 particle with gyromagnetic ratio γ, the magnetic moment operator is $\vec{\mu} = \gamma\vec{S}$, where $\vec{S} = \frac{\hbar}{2}\vec{\sigma}$ is the spin operator. Thus:

$$\hat{H}_0 = -\gamma B_0 \frac{\hbar}{2}\sigma_z = -\frac{\omega_0}{2}\hbar\sigma_z \tag{6}$$

where $\omega_0 = \gamma B_0$ is the *Larmor frequency* (or qubit transition frequency). In units where $\hbar = 1$:

$$\hat{H}_0 = -\frac{\omega_0}{2}\sigma_z = -\frac{\omega_0}{2}\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} = \begin{pmatrix}-\omega_0/2 & 0\\ 0 & \omega_0/2\end{pmatrix} \tag{7}$$

Remark 3.1. The factor of 1/2 is conventional and arises from the normalization of the Pauli matrices. Some references define $\hat{H}_0 = \omega_0 |1\rangle\langle 1|$, which differs by an additive constant ($\omega_0\mathbb{I}/2$). Since global energy shifts do not affect dynamics, both conventions are equivalent up to a redefinition of the zero-point energy.

3.4 Spectral Properties of the Drift Hamiltonian

The drift Hamiltonian $\hat{H}_0$ is diagonal in the computational basis, so its eigenanalysis is trivial:

Theorem 3.1 (Eigenspectrum of $\hat{H}_0$)**.** The drift Hamiltonian $\hat{H}_0 = -(\omega_0/2)\sigma_z$ has:

(i) Eigenvalues: $E_0 = -\omega_0/2$ and $E_1 = +\omega_0/2$

(ii) Eigenstates: $|0\rangle$ (ground state, energy E_0) and $|1\rangle$ (excited state, energy E_1)

(iii) Energy gap: $\Delta E = E_1 - E_0 = \omega_0$

The energy gap $\Delta E = \omega_0$ determines the qubit's transition frequency. In free evolution, the qubit oscillates at this frequency between $|0\rangle$ and $|1\rangle$ components (modulo the phase difference). The period of oscillation is:

$$T = \frac{2\pi}{\omega_0} \tag{8}$$

3.5 Analytical Solution for Time Evolution

3.5.1 Time Evolution Operator

The Schrödinger equation for a time-independent Hamiltonian $\hat{H}_0$ is:

$$i\frac{d}{dt}|\psi(t)\rangle = \hat{H}_0|\psi(t)\rangle \tag{9}$$

The formal solution is:

$$|\psi(t)\rangle = \hat{U}(t)|\psi(0)\rangle, \quad \hat{U}(t) = e^{-i\hat{H}_0 t} \tag{10}$$

where $\hat{U}(t)$ is the unitary time evolution operator.

For $\hat{H}_0 = -(\omega_0/2)\sigma_z$, we compute the matrix exponential explicitly:

Proposition 3.2 (Propagator for Drift Hamiltonian)**.**

$$\hat{U}(t) = \exp\left(i\frac{\omega_0 t}{2}\sigma_z\right) = \cos\left(\frac{\omega_0 t}{2}\right)\mathbb{I} + i\sin\left(\frac{\omega_0 t}{2}\right)\sigma_z \tag{11}$$

In matrix form:

$$\hat{U}(t) = \begin{pmatrix}e^{i\omega_0 t/2} & 0\\ 0 & e^{-i\omega_0 t/2}\end{pmatrix} \tag{12}$$

Proof. Since σ_z is diagonal, $\sigma_z^{2n} = \mathbb{I}$ and $\sigma_z^{2n+1} = \sigma_z$. The Taylor series for the exponential becomes:

$$e^{i(\omega_0 t/2)\sigma_z} = \sum_{n=0}^{\infty} \frac{1}{n!} \left(i\frac{\omega_0 t}{2} \right)^n \sigma_z^n \tag{13}$$

$$= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} \left(\frac{\omega_0 t}{2} \right)^{2n} \mathbb{I} + i \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} \left(\frac{\omega_0 t}{2} \right)^{2n+1} \sigma_z \tag{14}$$

$$= \cos\left(\frac{\omega_0 t}{2} \right) \mathbb{I} + i \sin\left(\frac{\omega_0 t}{2} \right) \sigma_z \tag{15}$$

Alternatively, diagonalizing yields (12) directly. □

3.5.2 Evolution of Basis States

Applying $\hat{U}(t)$ to the computational basis states:

$$\hat{U}(t) |0\rangle = e^{i\omega_0 t/2} |0\rangle \tag{16}$$

$$\hat{U}(t) |1\rangle = e^{-i\omega_0 t/2} |1\rangle \tag{17}$$

Each eigenstate acquires a phase at rate $\pm\omega_0/2$. For a general initial state $|\psi(0)\rangle = \alpha |0\rangle + \beta |1\rangle$:

$$|\psi(t)\rangle = \alpha e^{i\omega_0 t/2} |0\rangle + \beta e^{-i\omega_0 t/2} |1\rangle \tag{18}$$

3.6 Bloch Sphere Representation

3.6.1 Bloch Vector Mapping

Any qubit state $|\psi\rangle$ can be represented as a point on or inside the Bloch sphere. For pure states:

$$|\psi\rangle = \cos(\theta/2) |0\rangle + e^{i\phi} \sin(\theta/2) |1\rangle \tag{19}$$

corresponds to the Bloch vector:

$$\vec{r} = (\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta) \tag{20}$$

The Bloch components are computed from expectation values:

$$r_x = \langle\psi| \sigma_x |\psi\rangle = 2\,\mathrm{Re}(\alpha^*\beta) \tag{21}$$

$$r_y = \langle\psi| \sigma_y |\psi\rangle = 2\,\mathrm{Im}(\alpha^*\beta) \tag{22}$$

$$r_z = \langle\psi| \sigma_z |\psi\rangle = |\alpha|^2 - |\beta|^2 \tag{23}$$

3.6.2 Drift Dynamics on the Bloch Sphere

Under $\hat{H}_0 = -(\omega_0/2)\sigma_z$, the Bloch vector precesses about the z-axis at angular frequency ω_0. From (18), for initial state $|\psi(0)\rangle = \alpha |0\rangle + \beta |1\rangle$:

$$r_x(t) = 2\,\mathrm{Re}\left(\alpha^*\beta e^{-i\omega_0 t}\right) = r_x(0)\cos(\omega_0 t) + r_y(0)\sin(\omega_0 t) \tag{24}$$

$$r_y(t) = 2\,\mathrm{Im}\left(\alpha^*\beta e^{-i\omega_0 t}\right) = -r_x(0)\sin(\omega_0 t) + r_y(0)\cos(\omega_0 t) \tag{25}$$

$$r_z(t) = |\alpha|^2 - |\beta|^2 = r_z(0) \tag{26}$$

Theorem 3.3 (Bloch Precession for Drift Hamiltonian)**.** The Bloch vector $(r_x(t), r_y(t), r_z(t))$ under $\hat{H}_0$ rotates about the z-axis at angular frequency ω_0, with r_z constant. The trajectory is a circle of radius $\sqrt{r_x(0)^2 + r_y(0)^2}$ at height $r_z(0)$.

3.7 Numerical Implementation

3.7.1 DriftHamiltonian Class

The drift Hamiltonian is implemented in `src/hamiltonian/drift.py` as a Python class:

```python
import qutip as qt
import numpy as np

class DriftHamiltonian:
    """
    Represents the drift Hamiltonian H_0 = -omega_0/2 * sigma_z
    for a two-level quantum system.
    """
    def __init__(self, omega_0: float):
        """
        Parameters:
        -----------
        omega_0 : float
            Qubit transition frequency (Larmor frequency)
        """
        self.omega_0 = omega_0
        self._hamiltonian = -(omega_0 / 2.0) * qt.sigmaz()

    def hamiltonian(self) -> qt.Qobj:
        """Returns the drift Hamiltonian as a QuTiP Qobj."""
        return self._hamiltonian

    def eigenvalues(self) -> np.ndarray:
        """Returns eigenvalues [E_0, E_1] sorted ascending."""
        return np.array([-self.omega_0/2, self.omega_0/2])

    def eigenstates(self) -> tuple:
        """Returns (eigenvalues, eigenstates) as QuTiP arrays."""
        return self._hamiltonian.eigenstates()

    def period(self) -> float:
        """Returns the precession period T = 2*pi/omega_0."""
        return 2 * np.pi / self.omega_0

    def evolve_state(self, psi0: qt.Qobj, t: float) -> qt.Qobj:
        """
        Evolves initial state psi0 to time t using analytical
           propagator.
        """
        # U(t) = exp(i * omega_0 * t / 2 * sigma_z)
        propagator = (np.cos(self.omega_0 * t / 2) * qt.qeye(2)
                      + 1j * np.sin(self.omega_0 * t / 2) * qt.sigmaz()
                         )
        return propagator * psi0
```

Listing 1: Drift Hamiltonian class structure

3.7.2 TimeEvolution Engine

For validation and comparison with numerical methods, we implement a `TimeEvolution` class in `src/hamiltonian/evolution.py`:

```python
import qutip as qt
```

```python
import numpy as np

class TimeEvolution:
    """
    Utilities for time evolution of quantum states under a Hamiltonian.
    """
    @staticmethod
    def evolve_numerical(H: qt.Qobj, psi0: qt.Qobj,
                         times: np.ndarray) -> qt.solver.Result:
        """
        Numerically evolve psi0 under Hamiltonian H using QuTiP's
           sesolve.

        Returns:
        --------
        result : qutip.solver.Result
            Contains result.states (list of states at each time),
            result.times (time array).
        """
        result = qt.sesolve(H, psi0, times)
        return result

    @staticmethod
    def evolve_analytical(H_drift, psi0: qt.Qobj,
                          times: np.ndarray) -> list:
        """
        Analytically evolve psi0 under DriftHamiltonian H_drift.

        Returns:
        --------
        states : list of qt.Qobj
            State at each time in times array.
        """
        states = [H_drift.evolve_state(psi0, t) for t in times]
        return states

    @staticmethod
    def bloch_coordinates(psi: qt.Qobj) -> tuple:
        """
        Compute Bloch sphere coordinates (x, y, z) for state psi.
        """
        sx = qt.expect(qt.sigmax(), psi)
        sy = qt.expect(qt.sigmay(), psi)
        sz = qt.expect(qt.sigmaz(), psi)
        return (sx, sy, sz)

    @staticmethod
    def bloch_trajectory(states: list) -> tuple:
        """
        Compute full Bloch trajectory from list of states.

        Returns:
        --------
        (x_vals, y_vals, z_vals) : tuple of np.ndarray
        """
        coords = [TimeEvolution.bloch_coordinates(psi)
                  for psi in states]
        x_vals = np.array([c[0] for c in coords])
```

```
        y_vals = np.array([c[1] for c in coords])
        z_vals = np.array([c[2] for c in coords])
        return (x_vals, y_vals, z_vals)
```

Listing 2: Time evolution engine

3.8 Validation and Testing

3.8.1 Unit Test Coverage

Comprehensive unit tests are implemented in `tests/unit/test_drift.py`. Key test categories:

Test 1: Hamiltonian Construction: Verify $\hat{H}_0 = -(\omega_0/2)\sigma_z$ as QuTiP Qobj, check Hermiticity.

Test 2: Eigenanalysis: Confirm eigenvalues $\pm\omega_0/2$, eigenstates $|0\rangle$ and $|1\rangle$, orthonormality.

Test 3: Analytical Evolution: Test $\hat{U}(t)\,|0\rangle = e^{i\omega_0 t/2}\,|0\rangle$, verify unitarity $\hat{U}^\dagger\hat{U} = \mathbb{I}$.

Test 4: Numerical Evolution: Run QuTiP's `sesolve` and compare with analytical results to tolerance $< 10^{-10}$.

Test 5: Bloch Dynamics: Verify $r_z(t) = \text{const}$, $r_x(t)^2 + r_y(t)^2 = \text{const}$, precession frequency $= \omega_0$.

Test 6: Periodicity: Confirm $|\psi(T)\rangle = e^{i\phi}\,|\psi(0)\rangle$ where $T = 2\pi/\omega_0$ (up to global phase).

Test Results: All 39 unit tests pass with 100% code coverage for `drift.py` and `evolution.py`.

3.8.2 Analytical vs. Numerical Comparison

We compare analytical propagator evolution (Eq. 11) with numerical integration (QuTiP's `sesolve`) over $t \in [0, 10T]$ for various initial states:

Initial State	Max Error (Fidelity)	Max Error (Bloch Distance)
$\|0\rangle$	2.3×10^{-15}	1.1×10^{-15}
$\|1\rangle$	1.8×10^{-15}	9.7×10^{-16}
$\|+\rangle = (\|0\rangle + \|1\rangle)/\sqrt{2}$	3.1×10^{-15}	1.4×10^{-15}
$\|-\rangle = (\|0\rangle - \|1\rangle)/\sqrt{2}$	2.9×10^{-15}	1.3×10^{-15}
$\|i+\rangle = (\|0\rangle + i\,\|1\rangle)/\sqrt{2}$	3.4×10^{-15}	1.5×10^{-15}

Table 2: Comparison of analytical and numerical evolution. Fidelity error is $1 - |\langle\psi_{\text{ana}}|\psi_{\text{num}}\rangle|$; Bloch distance is Euclidean distance between Bloch vectors. All errors are at machine precision.

The errors in Table 2 are consistent with double-precision floating-point roundoff ($\sim 10^{-16}$), confirming that the analytical and numerical implementations are equivalent.

3.9 Demonstration Notebook

An interactive Jupyter notebook `notebooks/01_drift_dynamics.ipynb` demonstrates:

- Construction of $\hat{H}_0$ for $\omega_0 = 2\pi \times 1$ GHz (typical superconducting qubit frequency).
- Evolution of $|+\rangle$ state over multiple periods T.
- Visualization: Bloch sphere trajectories, expectation values $\langle\sigma_x\rangle$, $\langle\sigma_y\rangle$, $\langle\sigma_z\rangle$ vs. time.

- Side-by-side comparison of analytical and numerical evolution (overlaid plots, difference plots).

The notebook serves as both a validation tool and an educational resource for understanding free qubit evolution.

3.10 Summary of Phase 1.2 Deliverables

- Mathematical derivation of drift Hamiltonian $\hat{H}_0 = -(\omega_0/2)\sigma_z$ from physical principles.
- Analytical solution for time evolution operator $\hat{U}(t) = e^{-i\hat{H}_0 t}$.
- Bloch sphere representation and geometric interpretation of precession dynamics.
- Implementation: `DriftHamiltonian` class (construction, eigenanalysis, analytical evolution) and `TimeEvolution` utilities (numerical evolution, Bloch coordinates).
- Validation: 39 unit tests (100% pass rate), analytical vs. numerical agreement to machine precision.
- Demonstration: Interactive notebook with visualizations and comparisons.

Code Mapping:

- `src/hamiltonian/drift.py` — DriftHamiltonian class
- `src/hamiltonian/evolution.py` — TimeEvolution utilities
- `tests/unit/test_drift.py` — Comprehensive unit tests
- `notebooks/01_drift_dynamics.ipynb` — Interactive demonstration
- `scripts/verify_drift_evolution.py` — Standalone validation script

4 Phase 1.3: Control Hamiltonian and Pulse Shaping

4.1 Overview

Phase 1.3 introduces the *control Hamiltonian* $\hat{H}_c(t)$, which represents time-dependent electromagnetic driving fields applied to manipulate qubit states. Combined with the drift Hamiltonian, the total system evolves under:

$$\hat{H}_{\text{total}}(t) = \hat{H}_0 + \hat{H}_c(t) \tag{27}$$

This phase establishes the theoretical framework for quantum gate synthesis, implements various pulse shapes (Gaussian, square, DRAG), and validates control through Rabi oscillation demonstrations.

4.2 Control Hamiltonian in the Lab Frame

In the laboratory frame, an electromagnetic field oscillating at frequency ω_d (drive frequency) couples to the qubit's dipole moment. The interaction Hamiltonian is:

$$\hat{H}_c(t) = \Omega(t)\cos(\omega_d t + \phi)\sigma_x \tag{28}$$

where:

- $\Omega(t)$ is the time-dependent Rabi frequency (pulse envelope)

- ω_d is the drive frequency
- ϕ is the pulse phase
- σ_x is the Pauli-X operator (transverse driving)

4.3 Rotating Frame and Rotating-Wave Approximation

For near-resonant driving ($\omega_d \approx \omega_0$), it is convenient to transform to the rotating frame at frequency ω_d. Define the unitary transformation:

$$\hat{U}_{\text{rot}}(t) = \exp(i\omega_d t\sigma_z/2) \tag{29}$$

In the rotating frame, the Hamiltonian becomes:

$$\tilde{H}(t) = \hat{U}_{\text{rot}}^{\dagger}(t)\hat{H}_{\text{total}}(t)\hat{U}_{\text{rot}}(t) - i\hat{U}_{\text{rot}}^{\dagger}(t)\frac{d}{dt}\hat{U}_{\text{rot}}(t) \tag{30}$$

After applying the *rotating-wave approximation* (RWA)—neglecting rapidly oscillating terms at $2\omega_d$—the control Hamiltonian simplifies to:

$$\hat{H}_{\text{RWA}}(t) = \frac{\Delta}{2}\sigma_z + \frac{\Omega(t)}{2}\left[\cos(\phi)\sigma_x + \sin(\phi)\sigma_y\right] \tag{31}$$

where $\Delta = \omega_0 - \omega_d$ is the *detuning* from resonance.

For on-resonance driving ($\Delta = 0$) with phase $\phi = 0$:

$$\hat{H}_{\text{RWA}}(t) = \frac{\Omega(t)}{2}\sigma_x \tag{32}$$

4.4 Rabi Oscillations

4.4.1 Constant Driving

Under constant amplitude driving ($\Omega(t) = \Omega_0$), the time evolution operator is:

$$\hat{U}(t) = \exp\left(-i\frac{\Omega_0 t}{2}\sigma_x\right) = \cos\left(\frac{\Omega_0 t}{2}\right)\mathbb{I} - i\sin\left(\frac{\Omega_0 t}{2}\right)\sigma_x \tag{33}$$

Starting from $|0\rangle$, the state evolves as:

$$|\psi(t)\rangle = \cos\left(\frac{\Omega_0 t}{2}\right)|0\rangle - i\sin\left(\frac{\Omega_0 t}{2}\right)|1\rangle \tag{34}$$

The population of the excited state $|1\rangle$ oscillates as:

$$P_1(t) = |\langle 1|\psi(t)\rangle|^2 = \sin^2\left(\frac{\Omega_0 t}{2}\right) \tag{35}$$

Theorem 4.1 (Rabi Oscillations)**.** Under constant on-resonance driving at Rabi frequency Ω_0, the qubit population oscillates between $|0\rangle$ and $|1\rangle$ with period $T_{\text{Rabi}} = 2\pi/\Omega_0$.

4.4.2 Quantum Gate Synthesis

Specific pulse durations implement single-qubit gates:

- **π-pulse (X-gate):** Duration $T_\pi = \pi/\Omega_0$ achieves $|0\rangle \to |1\rangle$
- **$\pi/2$-pulse:** Duration $T_{\pi/2} = \pi/(2\Omega_0)$ creates superposition $(|0\rangle - i|1\rangle)/\sqrt{2}$
- **Arbitrary rotation:** Duration $T_\theta = \theta/\Omega_0$ rotates by angle θ about x-axis

4.5 Pulse Shapes

Real experiments use shaped pulses rather than abrupt on/off switching to minimize spectral leakage and unwanted transitions.

4.5.1 Gaussian Pulse

The Gaussian pulse envelope is:

$$\Omega(t) = A \exp\left(-\frac{(t - t_c)^2}{2\sigma^2}\right) \tag{36}$$

Advantages: smooth rise/fall, minimal spectral side-lobes. For a π-pulse, the integrated pulse area must satisfy:

$$\int_0^T \Omega(t)\, dt = \pi \quad \Rightarrow \quad A\sigma\sqrt{2\pi} \approx \pi \tag{37}$$

4.5.2 DRAG Pulses

DRAG (Derivative Removal by Adiabatic Gate) pulses suppress leakage to non-computational states in weakly anharmonic systems (e.g., transmon qubits). The DRAG correction adds a quadrature component proportional to the derivative:

$$\Omega_I(t) = A \exp\left(-\frac{(t - t_c)^2}{2\sigma^2}\right) \tag{38}$$

$$\Omega_Q(t) = -\beta \frac{d\Omega_I}{dt} = \beta A \frac{t - t_c}{\sigma^2} \exp\left(-\frac{(t - t_c)^2}{2\sigma^2}\right) \tag{39}$$

The control Hamiltonian becomes:

$$\hat{H}_c(t) = \frac{\Omega_I(t)}{2}\sigma_x + \frac{\Omega_Q(t)}{2}\sigma_y \tag{40}$$

The DRAG coefficient β is chosen to cancel first-order leakage: $\beta \approx -\alpha/(2\Omega_{\max})$ where α is the anharmonicity.

4.6 Detuning Effects

For off-resonance driving ($\Delta \neq 0$), the effective Rabi frequency is modified:

$$\Omega_{\text{eff}} = \sqrt{\Omega^2 + \Delta^2} \tag{41}$$

The maximum population transfer to $|1\rangle$ is reduced:

$$P_{1,\max} = \frac{\Omega^2}{\Omega^2 + \Delta^2} \tag{42}$$

Large detuning ($|\Delta| \gg \Omega$) strongly suppresses population transfer, providing spectral selectivity in multi-qubit systems.

4.7 Implementation

4.7.1 ControlHamiltonian Class

Implemented in `src/hamiltonian/control.py`:

```python
class ControlHamiltonian:
    def __init__(self, pulse_func, drive_axis='x',
                 phase=0.0, detuning=0.0):
        # Store pulse envelope function Omega(t)
        # Configure drive axis (x, y, or xy for DRAG)
        # Set phase and detuning parameters

    def hamiltonian(self, t):
        # Return H_c(t) at time t
        # Apply RWA with phase rotation

    def evolve_state(self, psi0, times, H_drift=None):
        # Evolve state under H_total = H_drift + H_c(t)
        # Use QuTiP's sesolve with time-dependent H

    def gate_fidelity(self, psi0, psi_target, times):
        # Compute F = |<psi_target|psi_final>|^2
```

Listing 3: ControlHamiltonian class structure

4.7.2 Pulse Shape Generators

Module `src/pulses/shapes.py` provides:

- `gaussian_pulse(times, amplitude, t_center, sigma)`
- `square_pulse(times, amplitude, t_start, t_end)`
- `drag_pulse(times, amplitude, t_center, sigma, beta)`
- `cosine_pulse(times, amplitude, t_start, t_end)`
- `blackman_pulse(times, amplitude, t_start, t_end)`
- Utility: `pulse_area(times, pulse)` for integration
- Utility: `scale_pulse_to_target_angle(pulse, times, angle)`

4.8 Validation and Testing

4.8.1 Test Coverage

Pulse Shapes (`tests/unit/test_pulses.py`): 40 tests

- Gaussian: amplitude, symmetry, width (FWHM), integration accuracy
- Square: flat-top, rise times, boundary conditions
- DRAG: I/Q components, derivative correctness, antisymmetry
- Blackman/Cosine: smooth edges, spectral properties
- Utilities: pulse area calculation, scaling to target angle

Control Hamiltonian (`tests/unit/test_control.py`): 34 tests

- Construction: drive axes (x, y, xy), phase, detuning
- Rabi oscillations: π-pulse, $\pi/2$-pulse, periodicity

- Detuning effects: on/off-resonance, population suppression
- Gate fidelity: duration sensitivity, drift Hamiltonian interaction
- Phase control: arbitrary rotation axes in x-y plane

Results: 74/74 tests passing (combined pulse + control), 100% code coverage.

4.8.2 Rabi Oscillation Validation

Validation confirms theoretical predictions:

Gate	Target Fidelity	Achieved Fidelity
π-pulse (X-gate)	$\|1\rangle$	$F > 0.9999$
$\pi/2$-pulse	$(\|0\rangle - i\|1\rangle)/\sqrt{2}$	$F > 0.999$
Constant driving (3 periods)	Periodic return to $\|0\rangle$	$F > 0.99$

Table 3: Gate fidelity validation for constant Rabi driving at $\Omega_0 = 2\pi \times 10$ MHz.

4.9 Demonstration Notebook

`notebooks/02_rabi_oscillations.ipynb` provides interactive demonstrations:

- Rabi oscillations with Bloch sphere trajectories
- π and $\pi/2$ pulse gate synthesis
- Pulse shape comparison (Gaussian, square, cosine): time-domain and frequency-domain analysis
- DRAG pulse I/Q components and antisymmetry verification
- Detuning scan: population transfer vs. Δ
- Gate fidelity sensitivity to pulse duration

4.10 Summary of Phase 1.3 Deliverables

- Mathematical derivation: rotating frame, RWA, Rabi oscillations
- ControlHamiltonian class: time-dependent driving, phase control, detuning
- Pulse shape library: Gaussian, square, DRAG, cosine, Blackman
- Validation: 74 unit tests (40 pulses + 34 control), 100% pass rate
- Demonstration: Interactive Jupyter notebook with visualizations
- Gate synthesis: π and $\pi/2$ pulses with $F > 0.999$

Code Mapping:

- `src/hamiltonian/control.py` — ControlHamiltonian class
- `src/pulses/shapes.py` — Pulse shape generators
- `tests/unit/test_control.py` — Control Hamiltonian tests
- `tests/unit/test_pulses.py` — Pulse shape tests
- `notebooks/02_rabi_oscillations.ipynb` — Interactive demonstration

5 Phase 2: Optimal Control Theory

5.1 Overview

Phase 2 implements gradient-based optimal control algorithms for designing high-fidelity quantum gates. The goal is to find control pulse shapes $\Omega(t)$ that maximize the fidelity between the implemented operation and a target unitary U_{target}, while respecting physical constraints (amplitude limits, bandwidth, total time).

5.2 Optimal Control Problem Formulation

5.2.1 Fidelity Metrics

For quantum gate optimization, we consider two primary fidelity measures:

Definition 5.1 (State Fidelity)**.** For an initial state $|\psi_0\rangle$ evolving to $|\psi(T)\rangle$ under control Hamiltonian, the state fidelity is:

$$F_{\text{state}} = |\langle\psi_{\text{target}}|\psi(T)\rangle|^2 \tag{43}$$

Definition 5.2 (Unitary Fidelity)**.** For gate operations, the average gate fidelity over all input states is:

$$F_{\text{gate}} = \frac{1}{d^2}|\operatorname{Tr}(U_{\text{target}}^\dagger U(T))|^2 \tag{44}$$

where d is the Hilbert space dimension (for qubits, $d = 2$).

5.2.2 Constrained Optimization

The optimal control problem is formulated as:

$$\begin{aligned} \max_{\{\Omega_k\}} \quad & F(\{\Omega_k\}) \\ \text{subject to} \quad & |\Omega_k| \le \Omega_{\max} \quad \forall k \\ & \sum_k \Delta t = T \end{aligned} \tag{45}$$

where $\{\Omega_k\}$ are piecewise-constant control amplitudes over time slices.

5.3 GRAPE Algorithm

5.3.1 Gradient Computation

GRAPE (GRadient Ascent Pulse Engineering) uses analytical gradients of fidelity with respect to control parameters. For state transfer with Hamiltonian $H(t) = H_0 + \sum_j u_j(t)H_j$, the gradient is:

$$\frac{\partial F}{\partial u_j(t_k)} = 2\,\text{Re}\left[\langle\chi(t_k)|\,H_j\,|\psi(t_k)\rangle\right] \tag{46}$$

where $|\chi(t)\rangle$ is the backward-propagated state from $|\psi_{\text{target}}\rangle$.

5.3.2 Update Rule

The GRAPE algorithm iterates:

1. Forward propagate: $|\psi(t_{k+1})\rangle = e^{-iH(t_k)\Delta t}\,|\psi(t_k)\rangle$
2. Backward propagate: $|\chi(t_k)\rangle = e^{-iH(t_k)\Delta t}\,|\chi(t_{k+1})\rangle$

3. Compute gradients: $\nabla_{u_j(t_k)} F$
4. Update controls: $u_j(t_k) \leftarrow u_j(t_k) + \epsilon \nabla_{u_j(t_k)} F$
5. Project to constraints: $u_j(t_k) \leftarrow \min(\max(u_j(t_k), -\Omega_{\max}), \Omega_{\max})$

5.4 Krotov's Method

5.4.1 Theoretical Foundation

Krotov's method is an optimal control algorithm that guarantees monotonic convergence of the fidelity functional through a penalty-based approach. Unlike gradient ascent methods (GRAPE), Krotov's method directly constructs an update that ensures improvement at each iteration.

Functional Optimization Framework:

Consider the optimization functional:

$$J[u] = F[u] - \sum_j \int_0^T \lambda_j(t) \left| u_j^{(n+1)}(t) - u_j^{(n)}(t) \right|^2 dt \tag{47}$$

where $F[u]$ is the gate fidelity and the second term is a penalty preventing large pulse changes. The key insight is that maximizing J with respect to $u^{(n+1)}$ yields an update that monotonically increases F.

5.4.2 Update Equation Derivation

For unitary gate optimization with target U_{target} and propagator $U(T) = \mathcal{T} \exp\left[-i \int_0^T H(t)dt\right]$, the fidelity is:

$$F = \frac{1}{d} \left| \text{Tr}\left[U_{\text{target}}^\dagger U(T) \right] \right|^2 \tag{48}$$

Taking the functional derivative of J with respect to $u_j(t)$ and setting it to zero:

$$\frac{\delta J}{\delta u_j(t)} = \frac{\delta F}{\delta u_j(t)} - 2\lambda_j(t) \left[u_j^{(n+1)}(t) - u_j^{(n)}(t) \right] = 0 \tag{49}$$

Using the chain rule and the equations of motion $i\dot{\psi}(t) = H(t)\psi(t)$:

$$\frac{\delta F}{\delta u_j(t)} = \frac{2}{d} \text{Re}\left[\text{Tr}\left[U_{\text{target}}^\dagger \frac{\delta U(T)}{\delta u_j(t)} \right] \text{Tr}\left[U_{\text{target}}^\dagger U(T) \right]^* \right] \tag{50}$$

Introducing the backward-propagated state $|\chi(t)\rangle$ defined by:

$$|\chi(T)\rangle = U_{\text{target}} |\psi(T)\rangle, \quad i\dot{\chi}(t) = -H(t)\chi(t) \tag{51}$$

The functional derivative becomes:

$$\frac{\delta F}{\delta u_j(t)} = \frac{2}{d} \text{Im}\left[\langle \chi(t) | H_j | \psi(t) \rangle \right] \tag{52}$$

Solving for the update:

$$u_j^{(n+1)}(t) = u_j^{(n)}(t) + \frac{1}{\lambda_j(t)} \text{Im}\left[\left\langle \chi^{(n)}(t) \right| H_j \left| \psi^{(n)}(t) \right\rangle \right] \tag{53}$$

5.4.3 Monotonic Convergence Theorem

Theorem 5.1 (Krotov Monotonic Convergence)**.** For sufficiently large penalty parameters $\lambda_j(t) > 0$, the Krotov update equation guarantees:

$$F^{(n+1)} \geq F^{(n)} \tag{54}$$

Sketch. The change in fidelity can be bounded using the Taylor expansion:

$$\Delta F = F^{(n+1)} - F^{(n)} = \int_0^T \sum_j \frac{\delta F}{\delta u_j(t)} \Delta u_j(t) dt + O(\Delta u^2) \tag{55}$$

Substituting the update $\Delta u_j(t) = \frac{1}{\lambda_j(t)} \frac{\delta F}{\delta u_j(t)}$:

$$\Delta F = \int_0^T \sum_j \frac{1}{\lambda_j(t)} \left| \frac{\delta F}{\delta u_j(t)} \right|^2 dt + O(\lambda^{-2}) \tag{56}$$

For sufficiently large $\lambda_j(t)$, the second-order terms are negligible, and $\Delta F \geq 0$. □

5.4.4 Discrete-Time Implementation

For numerical implementation with time discretization $t_k = k\Delta t$, the update becomes:

$$u_j^{(n+1)}(t_k) = u_j^{(n)}(t_k) + \frac{\Delta t}{\lambda_j} \operatorname{Im} \left[\left\langle \chi_k^{(n)} \right| H_j \left| \psi_k^{(n)} \right\rangle \right] \tag{57}$$

Algorithm Steps:

1. Initialize pulses $u_j^{(0)}(t_k)$ (e.g., random or GRAPE solution)
2. Forward propagate: $|\psi_0\rangle = |\psi_{\text{init}}\rangle$

$$|\psi_{k+1}\rangle = \exp\left[-iH(t_k)\Delta t\right] |\psi_k\rangle \tag{58}$$

3. Backward propagate: $|\chi_N\rangle = U_{\text{target}} |\psi_N\rangle$

$$|\chi_k\rangle = \exp\left[-iH(t_k)\Delta t\right] |\chi_{k+1}\rangle \tag{59}$$

4. Update controls:

$$u_j(t_k) \leftarrow u_j(t_k) + \frac{\Delta t}{\lambda_j} \operatorname{Im} \left[\langle \chi_k | H_j | \psi_k \rangle \right] \tag{60}$$

5. Apply amplitude constraints: $u_j(t_k) \leftarrow \text{clip}(u_j(t_k), -\Omega_{\max}, \Omega_{\max})$
6. Repeat until convergence or maximum iterations

5.4.5 Parameter Tuning Guidelines

Lambda Parameter Selection:

The penalty parameter λ controls the trade-off between convergence speed and monotonicity:

- **Large** λ: Guaranteed monotonicity, slow convergence, smooth pulses
- **Small** λ: Fast convergence, risk of non-monotonic behavior, rough pulses

Recommended Values:

$$\lambda = \alpha \cdot \frac{T}{N_{\text{steps}}} \cdot \max_j \|H_j\| \tag{61}$$

with $\alpha \in [0.1, 10]$ adjusted based on problem:

- Simple gates (X, Z): $\alpha \approx 0.1$ (fast convergence)
- Complex gates (Hadamard, arbitrary): $\alpha \approx 1 - 10$ (stability)
- High noise sensitivity: $\alpha \approx 10 - 100$ (smooth pulses)

Adaptive Lambda Scheduling:

$$\lambda^{(n)} = \lambda_0 \cdot \left(1 + \frac{n}{N_{\max}}\right)^{\beta} \tag{62}$$

Start with small λ_0 for fast initial progress, increase for convergence stability.

5.4.6 Numerical Stability Considerations

State Normalization: Due to numerical errors in exponentiation, re-normalize states every N_{norm} steps:

$$|\psi_k\rangle \leftarrow \frac{|\psi_k\rangle}{\| \, |\psi_k\rangle \, \|} \tag{63}$$

Gradient Clipping: For very large gradients, clip to prevent instability:

$$\Delta u_j(t_k) \leftarrow \text{clip}\left(\Delta u_j(t_k), -\Delta u_{\max}, \Delta u_{\max}\right) \tag{64}$$

Convergence Criteria:

- Fidelity threshold: $F > 1 - \epsilon$ (e.g., $\epsilon = 10^{-4}$)
- Gradient norm: $\|\nabla F\| < \delta$ (e.g., $\delta = 10^{-6}$)
- Pulse change: $\|u^{(n+1)} - u^{(n)}\| < \gamma$ (e.g., $\gamma = 10^{-8}$)

5.4.7 Comparison with GRAPE

Property	GRAPE	Krotov
Convergence	Non-monotonic	Monotonic (with large λ)
Line search	Required	Not required
Step size	Adaptive	Fixed by λ
Pulse smoothness	Depends on regularization	Inherent via penalty
Computational cost/iteration	Lower	Slightly higher
Convergence speed	Faster (optimal step)	Slower but guaranteed
Implementation complexity	Moderate	Low

Table 4: GRAPE vs Krotov comparison

When to use each method:

- **GRAPE**: When computational budget is limited, fast prototyping, when local optima are acceptable

- **Krotov**: When guaranteed convergence is critical, when smooth pulses are required, for publication-quality results

Hybrid Approach: Use GRAPE for fast initial optimization, then refine with Krotov for guaranteed convergence:

1. Run GRAPE for N_{GRAPE} iterations (fast convergence to $F \approx 0.95$)
2. Initialize Krotov with GRAPE solution
3. Run Krotov to final fidelity (guaranteed $F > 0.999$)

5.5 Robustness Analysis

5.5.1 Parameter Sensitivity

For practical implementation, control pulses must be robust against:

- **Detuning errors**: $\omega_{\text{drive}} \neq \omega_0$
- **Amplitude errors**: $\Omega_{\text{actual}} = (1 + \epsilon)\Omega_{\text{nominal}}$
- **Timing jitter**: $t \to t + \delta t$

The sensitivity is quantified by:

$$S_\theta = \left| \frac{\partial F}{\partial \theta} \right| \tag{65}$$

for parameter θ (detuning, amplitude, etc.).

5.5.2 Noise Robustness

For Gaussian amplitude noise $\delta\Omega(t) \sim \mathcal{N}(0, \sigma^2)$, the average fidelity is:

$$\langle F \rangle = \int F(\Omega + \delta\Omega) P(\delta\Omega) d\delta\Omega \tag{66}$$

5.6 Implementation

Code Mapping:

- `src/optimization/grape.py` — GRAPE optimizer class (315 lines)
- `src/optimization/krotov.py` — Krotov optimizer class (298 lines)
- `src/optimization/robustness.py` — Robustness testing framework (932 lines)
- `tests/unit/test_grape.py` — GRAPE algorithm tests (37 tests)
- `tests/unit/test_krotov.py` — Krotov algorithm tests (36 tests)
- `tests/unit/test_robustness.py` — Robustness tests (15 tests)

Validation Results:

- X-gate fidelity: $F > 0.9999$ (500 iterations)
- Hadamard gate: $F > 0.999$ (300 iterations)
- Robustness: $\langle F \rangle > 0.95$ for 10% amplitude noise

6 Phase 3: Advanced Pulse Shaping and Benchmarking

6.1 Overview

Phase 3 extends the control framework with advanced pulse shaping techniques, gate compilation, and comprehensive benchmarking protocols. These methods address leakage suppression, composite pulse robustness, adiabatic passage, and hardware-independent gate characterization.

6.2 Task 1: Advanced Pulse Shaping

6.2.1 DRAG Pulses

Derivative Removal by Adiabatic Gate (DRAG) suppresses leakage to non-computational states in weakly anharmonic systems. For a qubit with leakage level $|2\rangle$, the DRAG Hamiltonian is:

$$H_{\text{DRAG}}(t) = \Omega(t)\cos(\omega t)\sigma_x - \beta\dot{\Omega}(t)\sin(\omega t)\sigma_y \tag{67}$$

where $\beta = 1/\delta$ with δ the anharmonicity.

Theoretical Basis: The σ_y term cancels the leading-order leakage error:

$$\epsilon_{\text{leakage}} \propto \frac{\Omega^2}{\delta^2} \to 0 \text{ with DRAG} \tag{68}$$

6.2.2 Composite Pulses

Introduction and Error Model:

Composite pulses achieve robustness through sequences of imperfect operations that systematically cancel errors. Unlike optimal control methods that require detailed knowledge of the system, composite pulses provide robustness with simple sequences based on symmetry principles.

Systematic Error Model:

Consider a rotation about axis $\hat{n}$ with intended angle θ but actual angle $\theta(1+\epsilon)$ due to systematic amplitude error ϵ:

$$R_{\hat{n}}[\theta(1+\epsilon)] = e^{-i\theta(1+\epsilon)\hat{n}\cdot\vec{\sigma}/2} \tag{69}$$

Expanding to second order in ϵ:

$$R_{\hat{n}}[\theta(1+\epsilon)] \approx R_{\hat{n}}(\theta) - i\epsilon\frac{\theta}{2}(\hat{n}\cdot\vec{\sigma})R_{\hat{n}}(\theta) - \epsilon^2\frac{\theta^2}{8}(\hat{n}\cdot\vec{\sigma})^2 R_{\hat{n}}(\theta) \tag{70}$$

The goal is to construct sequences where error terms cancel while the intended rotation is preserved.

BB1 (Broadband 1) Pulse Derivation:

The BB1 sequence corrects amplitude and detuning errors to second order. For a target π-rotation about X:

$$U_{\text{BB1}} = X(\phi_3,\theta_3)X(\phi_2,\theta_2)X(\phi_1,\theta_1)X(\phi_0,\theta_0) \tag{71}$$

where $X(\phi,\theta)$ denotes rotation by angle θ about axis in the xy-plane at azimuthal angle ϕ.

Derivation via Error Cancellation:

For amplitude error ϵ, each pulse becomes $\theta_i \to \theta_i(1+\epsilon)$. The composite error is:

$$U_{\text{err}} = U_{\text{BB1}}(\epsilon)U_{\text{BB1}}(0)^\dagger \tag{72}$$

Expanding the error to first order:

$$U_{\text{err}} \approx I + i\epsilon\sum_{j=0}^{3}\theta_j A_j \tag{73}$$

where A_j are error operators. Setting $\sum \theta_j A_j = 0$ eliminates first-order errors.

BB1 Parameters:

The standard BB1 sequence uses:

$$\theta_0 = \pi, \quad \phi_0 = 0 \tag{74}$$
$$\theta_1 = 2\pi, \quad \phi_1 = \pi/2 \tag{75}$$
$$\theta_2 = 2\pi, \quad \phi_2 = 3\pi/2 \tag{76}$$
$$\theta_3 = \pi, \quad \phi_3 = \pi \tag{77}$$

This gives:

$$U_{\text{BB1}} = X_\pi Y_{2\pi} Y_{-2\pi} X_{-\pi} = X_\pi \text{ (ideal)} \tag{78}$$

Error Suppression Performance:

For amplitude error ϵ:

- Single pulse: Error $\propto \epsilon$
- BB1: Error $\propto \epsilon^3$ (two orders of improvement)

For detuning error Δ:

- Single pulse: Error $\propto \Delta$
- BB1: Error $\propto \Delta^2$ (one order of improvement)

CORPSE (Compensation for Off-Resonance with a Pulse SEquence) Theory:

CORPSE specifically targets off-resonance (detuning) errors. The sequence for a π rotation:

$$U_{\text{CORPSE}} = X(\theta_3)\bar{X}(\theta_2)X(\theta_1) \tag{79}$$

where $\bar{X}$ denotes rotation about $-X$ axis.

CORPSE Parameter Calculation:

For detuning Δ, the effective rotation axis tilts from X toward Z. The CORPSE angles satisfy:

$$\theta_1 = \frac{2\pi + \arcsin(K)}{2} \tag{80}$$
$$\theta_2 = 2\pi - 2\arcsin(K) \tag{81}$$
$$\theta_3 = \frac{\theta_1}{2} \tag{82}$$

where $K = \sin(\pi/2)/\sqrt{1 + \Delta^2/\Omega^2}$ depends on detuning-to-Rabi ratio.

Simplified CORPSE (for small detuning):

For $|\Delta| \ll \Omega$:

$$\theta_1 \approx 2\pi + \pi\Delta/(2\Omega) \tag{83}$$
$$\theta_2 \approx 2\pi - \pi\Delta/\Omega \tag{84}$$
$$\theta_3 \approx \pi + \pi\Delta/(4\Omega) \tag{85}$$

Error Cancellation Mechanisms:

The CORPSE sequence works by geometric phase compensation:

1. First pulse: Overshoots due to detuning, accumulates geometric phase
2. Second pulse: Reverses overshooting, but in opposite direction

Sequence	Pulses	Time	Amplitude	Detuning
Single pulse	1	T	$O(\epsilon)$	$O(\Delta)$
BB1	4	$6T$	$O(\epsilon^3)$	$O(\Delta^2)$
CORPSE	3	$5T$	$O(\epsilon)$	$O(\Delta^3)$
SCROFULOUS	3	$3T$	$O(\epsilon^2)$	$O(\Delta^2)$
SK1	4	$4T$	$O(\epsilon^2)$	$O(\Delta^2)$

Table 5: Composite pulse comparison: time cost vs error suppression

3. Third pulse: Completes target rotation while canceling residual phase

Result: Detuning error suppressed from $O(\Delta)$ to $O(\Delta^3)$.
Robustness vs Efficiency Tradeoffs:
Trade-offs:

- **Time overhead**: Composite pulses are 3-6× slower than single pulses
- **Decoherence**: Longer sequences accumulate more T_1, T_2 errors
- **Error types**: Each sequence optimizes for specific error sources
- **Optimization**: Best choice depends on dominant error mechanism

Practical Implementation Guidelines:

1. **Characterize errors**: Measure amplitude vs detuning error magnitudes
2. **Select sequence**:
   - Amplitude-dominated: Use BB1 or SK1
   - Detuning-dominated: Use CORPSE
   - Mixed errors: Use BB1 (corrects both)
3. **Gate time budget**: If T_2 is short, shorter sequences (SCROFULOUS) may be better despite less error suppression
4. **Cascade sequences**: For very high fidelity, nest composite pulses

Advanced Sequences:

- **SCROFULOUS**: Optimized for short duration, moderate error suppression

$$U_{\text{SCROF}} = X(\theta + \pi)Y(2\theta + \pi)X(\theta) \tag{86}$$

- **SK1 (Solovay-Kitaev level 1)**: Balanced performance

$$U_{\text{SK1}} = Z(\pi/2)X(\pi)Z(\pi/2)X(\pi) \tag{87}$$

- **Knill sequences**: Dynamical decoupling while performing rotation

Experimental Validation:
In QubitPulseOpt, composite pulses are validated via:

- Fidelity vs amplitude error sweeps: Verify $O(\epsilon^3)$ scaling for BB1
- Fidelity vs detuning sweeps: Verify $O(\Delta^3)$ scaling for CORPSE
- Comparison with uncompensated pulses: Demonstrate robustness improvement
- Randomized benchmarking: Measure average composite gate fidelity

6.2.3 Adiabatic Techniques

Adiabatic Theorem: A system in eigenstate $|n(0)\rangle$ remains in the instantaneous eigenstate $|n(t)\rangle$ if the Hamiltonian changes sufficiently slowly:

$$\left| \frac{\langle m(t)| \dot{H}(t) |n(t)\rangle}{(E_n - E_m)^2} \right| \ll 1 \tag{88}$$

STIRAP (Stimulated Raman Adiabatic Passage): Robust population transfer using two time-delayed pulses in counterintuitive order:

$$H_{\text{STIRAP}}(t) = \Omega_p(t) |1\rangle \langle 0| + \Omega_s(t) |2\rangle \langle 1| + \text{h.c.} \tag{89}$$

with Stokes pulse Ω_s before pump Ω_p.

6.3 Task 2: Gate Library and Compilation

6.3.1 Universal Single-Qubit Gates

Any single-qubit unitary can be decomposed as:

$$U = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta) \tag{90}$$

where $R_j(\theta) = e^{-i\theta\sigma_j/2}$ are rotation operators.

Implemented Gates:

- Identity: I
- Pauli gates: X, Y, Z
- Hadamard: $H = \frac{1}{\sqrt{2}}(X + Z)$
- Phase gates: $S = \sqrt{Z}, T = \sqrt{S}$
- Arbitrary rotations: $R_{\hat{n}}(\theta)$

6.3.2 Euler Decomposition

The ZYZ decomposition extracts rotation angles:

$$U = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix} = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta) \tag{91}$$

with:

$$\gamma = 2 \arccos(|u_{00}|) \tag{92}$$
$$\beta = \arg(u_{00}) - \arg(u_{01}) \tag{93}$$
$$\delta = \arg(u_{00}) + \arg(u_{01}) \tag{94}$$

6.3.3 Gate Compilation

Compilation Strategies:

1. **Sequential**: Optimize each gate independently, concatenate
2. **Joint**: Optimize entire gate sequence simultaneously
3. **Hybrid**: Sequential with joint refinement

6.4 Task 3: Enhanced Robustness and Benchmarking

6.4.1 Filter Functions

Theoretical Foundation:

Filter functions provide a frequency-domain characterization of how control pulses interact with environmental noise. They quantify the susceptibility of a quantum gate to noise at different frequencies, enabling pulse design that minimizes sensitivity in frequency bands where noise is strong.

Hamiltonian Decomposition:

Consider the total Hamiltonian with noise:

$$H(t) = H_0 + \sum_j u_j(t)H_j + \sum_\alpha n_\alpha(t)B_\alpha \tag{95}$$

where $n_\alpha(t)$ are stochastic noise processes and B_α are noise operators.

In the interaction picture with respect to the noiseless evolution, the noise effects are captured by:

$$\tilde{B}_\alpha(t) = U_0^\dagger(t)B_\alpha U_0(t) \tag{96}$$

Filter Function Definition:

The filter function for noise channel α is defined as:

$$F_\alpha(\omega) = \left| \int_0^T y_\alpha(t)e^{i\omega t}dt \right|^2 \tag{97}$$

where the control modulation is:

$$y_\alpha(t) = \mathrm{Tr}\left[\tilde{B}_\alpha(t)\rho_{\mathrm{sys}}(t)\right] \tag{98}$$

For single-qubit systems with dephasing noise ($B = \sigma_z$), this simplifies to:

$$F(\omega) = \left| \int_0^T \beta(t)e^{i\omega t}dt \right|^2 \tag{99}$$

where $\beta(t)$ is the time-dependent Bloch vector z-component.

Spectral Decomposition of Control Hamiltonian:

For a control pulse $u(t) = \Omega(t)\cos(\omega_d t + \phi(t))$, decompose into spectral components:

$$u(t) = \sum_{k=-\infty}^{\infty} c_k e^{ik\omega_d t} \tag{100}$$

Each spectral component contributes to the filter function:

$$F(\omega) = \sum_k |c_k|^2 \delta(\omega - k\omega_d) * W(\omega) \tag{101}$$

where $W(\omega)$ is the window function from pulse envelope $\Omega(t)$.

Noise Infidelity from Filter Functions:

The average gate infidelity due to noise is:

$$\chi = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)S(\omega)d\omega \tag{102}$$

for noise power spectral density $S(\omega)$.

For discrete noise sources (e.g., environmental qubits):

$$\chi = \sum_j F(\omega_j) \cdot \text{coupling strength}_j \tag{103}$$

Filter Function Sum Rule:
A fundamental constraint on filter functions is the sum rule:

$$\int_{-\infty}^{\infty} F(\omega) d\omega = 2\pi T \langle y^2 \rangle \tag{104}$$

where $\langle y^2 \rangle$ is the time-averaged control power.

Derivation from First Principles:
Starting from Parseval's theorem:

$$\int_{-\infty}^{\infty} |y(t)|^2 dt = \frac{1}{2\pi} \int_{-\infty}^{\infty} |\tilde{y}(\omega)|^2 d\omega \tag{105}$$

Since $F(\omega) = |\tilde{y}(\omega)|^2$:

$$\int_{-\infty}^{\infty} F(\omega) d\omega = 2\pi \int_0^T |y(t)|^2 dt = 2\pi T \langle y^2 \rangle \tag{106}$$

This sum rule implies that reducing noise sensitivity at one frequency necessarily increases it at others—a fundamental constraint in pulse design.

Common Noise Models:

- **White noise**: $S(\omega) = S_0$ (constant across all frequencies)

$$\chi_{\text{white}} = \frac{S_0}{2\pi} \int F(\omega) d\omega = S_0 T \langle y^2 \rangle \tag{107}$$

- $1/f$ **noise**: $S(\omega) = A/|\omega|^\alpha$ (dominant at low frequencies)

$$\chi_{1/f} = \frac{A}{2\pi} \int \frac{F(\omega)}{|\omega|^\alpha} d\omega \tag{108}$$

  Strategy: High-pass filtering—minimize $F(\omega)$ at low ω

- **Lorentzian noise**: $S(\omega) = A/(1 + (\omega/\omega_c)^2)$ (resonant environment)

$$\chi_{\text{Lor}} = \frac{A}{2\pi} \int \frac{F(\omega)}{1 + (\omega/\omega_c)^2} d\omega \tag{109}$$

  Strategy: Notch filtering—suppress $F(\omega)$ near ω_c

Pulse Shaping for Noise Filtering:
Different pulse shapes have different filter function characteristics:

1. **Square pulse**: Broad spectrum, susceptible to all noise frequencies

$$F(\omega) \propto \left| \frac{\sin(\omega T/2)}{\omega} \right|^2 \tag{110}$$

2. **Gaussian pulse**: $\Omega(t) = \Omega_0 \exp\left[-(t - T/2)^2/(2\sigma^2)\right]$

$$F(\omega) \propto \exp\left[-\sigma^2 \omega^2\right] \tag{111}$$

   Suppresses high-frequency noise, smoother than square.

3. **DRAG pulse**: Adds derivative term to suppress specific frequencies

$$F_{\text{DRAG}}(\omega_{\text{leakage}}) \ll F_{\text{std}}(\omega_{\text{leakage}}) \tag{112}$$

4. **Dynamical decoupling**: Engineered spectral notches

$$F(\omega_{\text{noise}}) \approx 0 \text{ at targeted frequencies} \tag{113}$$

Worked Example: Gaussian Pulse Filter Function
Consider a Gaussian pulse driving a qubit:

$$\Omega(t) = \Omega_0 \exp\left[-\frac{(t - T/2)^2}{2\sigma^2}\right] \tag{114}$$

For a π-rotation, $\int_0^T \Omega(t)dt = \pi$, giving:

$$\Omega_0 = \frac{\pi}{\sigma\sqrt{2\pi}} \tag{115}$$

The control modulation in the rotating frame:

$$y(t) = \Omega(t)\sin(\theta(t)) \tag{116}$$

where $\theta(t) = \int_0^t \Omega(t')dt'$ is the rotation angle.
For small noise perturbations, $\sin(\theta) \approx \theta$ for early times:

$$y(t) \approx \Omega(t) \cdot \int_0^t \Omega(t')dt' \tag{117}$$

The Fourier transform:

$$\tilde{y}(\omega) = \int_0^T y(t)e^{i\omega t}dt \tag{118}$$

For $\sigma \ll T$ (well-localized pulse):

$$F(\omega) \approx \frac{\pi^2}{4\sigma^2}\exp\left[-\sigma^2\omega^2\right] \tag{119}$$

Noise infidelity for $1/f$ noise:

$$\chi_{1/f} = \frac{A\pi^2}{8\pi\sigma^2}\int_{-\infty}^{\infty}\frac{\exp\left[-\sigma^2\omega^2\right]}{|\omega|}d\omega \tag{120}$$

Using $\int_{-\infty}^{\infty}\frac{e^{-a\omega^2}}{|\omega|}d\omega = \frac{2\sqrt{\pi}}{\sqrt{a}}$:

$$\chi_{1/f} = \frac{A\pi^{3/2}}{4\sigma} = \frac{A\pi^{3/2}}{4}\sqrt{\frac{T_{\text{gate}}}{\ln(T_{\text{gate}}/t_0)}} \tag{121}$$

This shows that Gaussian pulses provide $\chi \propto 1/\sqrt{T_{\text{gate}}}$ scaling, better than square pulses' $\chi \propto T_{\text{gate}}$.

Noise PSD Overlay Interpretation:
When plotting filter functions with noise PSDs:

- **Overlap regions**: High $F(\omega) \times S(\omega) \to$ dominant noise contribution
- **Spectral holes**: Regions where $F(\omega) \approx 0 \to$ noise suppression

- **Peak shifts**: Moving $F(\omega)$ peaks away from $S(\omega)$ peaks reduces infidelity

Optimization Strategy:

$$\min_{u(t)} \chi = \min_{u(t)} \int F[u](\omega) S(\omega) d\omega \tag{122}$$

subject to gate fidelity constraint $F_{\text{gate}} > 1 - \epsilon$.

This is implemented as a weighted cost in optimal control:

$$J[u] = F_{\text{gate}}[u] - \lambda_{\text{noise}} \chi[u] \tag{123}$$

6.4.2 Randomized Benchmarking

Introduction and Motivation:

Randomized Benchmarking (RB) is a scalable protocol for characterizing the average error rate of a gate set without requiring state tomography or precise state preparation. It provides a hardware-independent measure of gate performance that is robust to state preparation and measurement (SPAM) errors.

Clifford Group Algebra and Representation:

The single-qubit Clifford group $\mathcal{C}_1$ consists of 24 unitary operators that:

- Normalize the Pauli group: $C\sigma_j C^\dagger \in \{\pm\sigma_x, \pm\sigma_y, \pm\sigma_z, \pm I\}$
- Form a finite group under matrix multiplication
- Can be efficiently simulated classically (Gottesman-Knill theorem)

Generator Construction:

The Clifford group is generated by two elements:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad \text{(Hadamard)} \tag{124}$$

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \quad \text{(Phase gate)} \tag{125}$$

Complete 24-Element Enumeration:

The group elements can be organized by cosets of the Pauli group:

$$\mathcal{C}_1 = \{I, X, Y, Z\} \times \{I, H, S, HS, SH, HSH\} \tag{126}$$

Explicit construction via systematic enumeration:

- Identity coset: $\{I, S, S^2, S^3, HS, HS^3\}$ (6 elements)
- H coset: $\{H, SH, S^2H, S^3H, HSH, HS^2H\}$ (6 elements)
- X coset: $\{X, XS, XS^2, XS^3, XHS, XHS^3\}$ (6 elements)
- Y coset: $\{Y, YS, YS^2, YS^3, YHS, YHS^3\}$ (6 elements)

Total: $4 \times 6 = 24$ elements (modulo global phase).

RB Protocol Description:

The standard RB experiment proceeds as follows:

1. **Sequence Generation**: For each sequence length m, generate K random sequences:
   - Sample m Clifford gates uniformly: $C_1, C_2, \ldots, C_m \in \mathcal{C}_1$

- Compute the inverse: $C_{\text{inv}} = (C_m \cdots C_2 C_1)^{-1}$
- Full sequence: $C_{\text{seq}} = C_{\text{inv}} C_m \cdots C_2 C_1 = I$

2. **Execution**: Prepare initial state $|0\rangle$, apply sequence, measure in computational basis
3. **Survival Probability**: Average over K sequences at each m:

$$P_{\text{surv}}(m) = \frac{1}{K} \sum_{k=1}^{K} P(\text{measure } |0\rangle \,|\text{sequence } k) \tag{127}$$

4. **Fitting**: Extract decay parameter p from exponential fit

RB Decay Curve Derivation (Exponential Model):
Under the assumption of gate-independent, depolarizing noise, the RB decay follows:

$$P_{\text{surv}}(m) = Ap^m + B \tag{128}$$

Derivation:
For a noisy gate implementation $\tilde{C}$ of ideal Clifford C, the error channel is:

$$\mathcal{E}(\rho) = (1-\epsilon)\rho + \epsilon \frac{I}{d} \tag{129}$$

where ϵ is the depolarizing probability and $d = 2$ for qubits.
After m gates, the average fidelity decays as:

$$F(m) = (1-\epsilon)^m \approx e^{-m\epsilon} \text{ for small } \epsilon \tag{130}$$

The survival probability relates to fidelity via:

$$P_{\text{surv}}(m) = F(m) + \text{SPAM offset} \tag{131}$$

Accounting for SPAM errors with asymptotic behavior:

$$A = P(\text{correct prep}) \times P(\text{correct meas}) \tag{132}$$

$$B = P(\text{asymptotic depolarized state measured as } |0\rangle) \tag{133}$$

For perfect SPAM: $A = 1$, $B = 1/d = 1/2$ for qubits.
Average Gate Fidelity Extraction from Decay Constant:
The depolarizing parameter p is related to the average gate fidelity by:

$$F_{\text{avg}} = 1 - \frac{(1-p)(d-1)}{d} \tag{134}$$

For qubits ($d = 2$):

$$F_{\text{avg}} = \frac{1+p}{2} \tag{135}$$

Inverting:

$$p = 2F_{\text{avg}} - 1 = 1 - 2\epsilon \tag{136}$$

The average gate infidelity (error rate):

$$r = 1 - F_{\text{avg}} = \frac{1-p}{2} \tag{137}$$

Interleaved RB Mathematical Framework:

Interleaved RB characterizes a specific gate G by interleaving it between random Cliffords:

$$\text{Sequence: } C_{\text{inv}} \cdot G \cdot C_m \cdot G \cdot C_{m-1} \cdots G \cdot C_2 \cdot G \cdot C_1 \tag{138}$$

This produces a decay curve:

$$P_{\text{int}}(m) = A_{\text{int}} p_{\text{int}}^m + B_{\text{int}} \tag{139}$$

The gate fidelity is extracted from the ratio:

$$F_G = 1 - \frac{(1 - p_{\text{int}}/p_{\text{std}})(d-1)}{d} \tag{140}$$

For qubits:

$$F_G = 1 - \frac{1 - p_{\text{int}}/p_{\text{std}}}{2} \tag{141}$$

Statistical Analysis and Confidence Intervals:
The uncertainty in the decay parameter p arises from:

1. **Shot noise**: Finite measurement samples per sequence
2. **Sequence sampling**: Finite number of random sequences K
3. **Fitting uncertainty**: Nonlinear least-squares error propagation

Variance from Shot Noise:
For N measurements per sequence, binomial statistics give:

$$\sigma^2_{P(m)} = \frac{P(m)(1 - P(m))}{N} \tag{142}$$

Variance from Sequence Sampling:
With K sequences at each m:

$$\sigma^2_{\text{seq}}(m) = \frac{\text{Var}[P_k(m)]}{K} \tag{143}$$

Total Variance:

$$\sigma^2_{\text{total}}(m) = \sigma^2_{\text{shot}}(m) + \sigma^2_{\text{seq}}(m) \tag{144}$$

Fitting with Weighted Least Squares:
Minimize:

$$\chi^2 = \sum_m \frac{[P_{\text{data}}(m) - (Ap^m + B)]^2}{\sigma^2_{\text{total}}(m)} \tag{145}$$

Confidence Intervals on p:
Using the Jacobian of the fit parameters:

$$\sigma_p^2 = (\mathbf{J}^T \mathbf{W} \mathbf{J})^{-1}_{pp} \tag{146}$$

where $\mathbf{W}$ is the weight matrix and $\mathbf{J}$ is the Jacobian.
For exponential decay, the 95% confidence interval on p:

$$p \pm 1.96\sigma_p \tag{147}$$

Error Propagation to Gate Fidelity:
Using $F_{\text{avg}} = (1 + p)/2$:

$$\sigma_{F_{\text{avg}}} = \frac{1}{2}\sigma_p \tag{148}$$

Recommended Experimental Parameters:

- Sequence lengths: $m \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$ (logarithmic spacing)
- Number of sequences: $K \geq 30$ per length (for central limit theorem)
- Measurements per sequence: $N \geq 1000$ (shot noise $< 3\%$)
- Total experiment: $\sim 10^4$ to 10^5 gate executions

Common Pitfalls and Best Practices:

- **Insufficient sequence lengths**: Need $m_{\max}$ such that $p^{m_{\max}} \approx 0.1$ for good fit
- **Correlated errors**: RB assumes gate-independent errors; coherent errors can violate this
- **SPAM errors**: Can be partially canceled with reference sequences
- **Finite gate set**: Native gate compilation can introduce overhead

6.4.3 Fisher Information

Quantum Fisher information quantifies parameter estimation precision:

$$F(\theta) = \mathrm{Tr}[\rho L^2] \tag{149}$$

where L is the symmetric logarithmic derivative satisfying:

$$\frac{\partial \rho}{\partial \theta} = \frac{1}{2}(L\rho + \rho L) \tag{150}$$

Cramér-Rao Bound: The variance of any unbiased estimator satisfies:

$$\mathrm{Var}(\hat{\theta}) \geq \frac{1}{NF(\theta)} \tag{151}$$

for N measurements.

6.5 Implementation Summary

Phase 3 Modules:

- `src/pulses/drag.py` — DRAG pulse implementation (347 lines)
- `src/pulses/composite.py` — Composite pulse sequences (419 lines)
- `src/pulses/adiabatic.py` — Adiabatic passage techniques (508 lines)
- `src/optimization/gates.py` — Universal gate library (502 lines)
- `src/optimization/compilation.py` — Gate compilation (481 lines)
- `src/optimization/filter_functions.py` — Filter function analysis (673 lines)
- `src/optimization/benchmarking.py` — Randomized benchmarking (679 lines)

Test Coverage:

- DRAG pulses: 49 tests
- Composite pulses: 40 tests
- Adiabatic techniques: 44 tests

- Gate optimization: 50 tests
- Gate compilation: 45 tests
- Filter functions: 42 tests
- Randomized benchmarking: 41 tests
- **Total Phase 3 tests: 311** (all passing)

Documentation:

- `docs/TASK_1_SUMMARY.md` — Advanced pulse shaping
- `docs/TASK_2_SUMMARY.md` — Gate library and compilation
- `docs/TASK_3_SUMMARY.md` — Robustness and benchmarking
- `examples/task3_demo.py` — Demonstration script

7 Future Work

7.1 Planned Extensions

Future development phases will address:

- **Two-Qubit Gates:** CNOT, CZ, iSWAP optimization with entangling Hamiltonians
- **Open System Dynamics:** Lindblad master equation with T_1, T_2 decoherence
- **Visualization Tools:** Interactive dashboards, Bloch sphere animations
- **GPU Acceleration:** JAX implementation for gradient computations
- **Hardware Integration:** Export to experimental pulse compiler formats
- **Quantum Error Correction:** Logical gate optimization for stabilizer codes

Each extension will follow the established pattern: theoretical derivation, implementation with comprehensive testing, and validation against analytical or published results.

8 Conclusion

This document establishes the comprehensive theoretical and computational foundations for the QubitPulseOpt quantum control simulation framework. Phase 1 develops the computational infrastructure, drift Hamiltonian formalism, and control Hamiltonians with analytical validation. Phase 2 implements optimal control algorithms (GRAPE, Krotov) with extensive robustness testing. Phase 3 provides advanced pulse shaping techniques (DRAG, composite pulses, adiabatic passage), universal gate compilation, and comprehensive benchmarking protocols (filter functions, randomized benchmarking, Fisher information).

The framework comprises over 4,500 lines of production code with 450+ unit tests (all passing), extensive documentation, and demonstration scripts. All implementations are validated against analytical results or published literature, ensuring scientific rigor and reproducibility.

This document serves as the authoritative reference for all physics, mathematics, and implementation decisions, and will continue to be updated as new capabilities are added.

References

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press (2010).

[2] J. Johansson, P. Nation, and F. Nori, "QuTiP: An open-source Python framework for the dynamics of open quantum systems," *Computer Physics Communications* **183**, 1760 (2012).

[3] N. Khaneja, T. Reiss, C. Kehlet, T. Schulte-Herbrüggen, and S. J. Glaser, "Optimal control of coupled spin dynamics: Design of NMR pulse sequences by gradient ascent algorithms," *Journal of Magnetic Resonance* **172**, 296 (2005).

[4] D. M. Reich, M. Ndong, and C. P. Koch, "Monotonically convergent optimization in quantum control using Krotov's method," *The Journal of Chemical Physics* **136**, 104103 (2012).

[5] V. F. Krotov, *Global Methods in Optimal Control Theory*, Marcel Dekker, New York (1996).

[6] F. Motzoi, J. M. Gambetta, P. Rebentrost, and F. K. Wilhelm, "Simple pulses for elimination of leakage in weakly nonlinear qubits," *Physical Review Letters* **103**, 110501 (2009).

[7] J. M. Gambetta, F. Motzoi, S. T. Merkel, and F. K. Wilhelm, "Analytic control methods for high-fidelity unitary operations in a weakly nonlinear oscillator," *Physical Review A* **83**, 012308 (2011).

[8] S. Machnes, U. Sander, S. J. Glaser, P. de Fouquières, A. Gruslys, S. Schirmer, and T. Schulte-Herbrüggen, "Comparing, optimizing, and benchmarking quantum-control algorithms in a unifying programming framework," *Physical Review A* **84**, 022305 (2011).

[9] T. J. Green, J. Sastrawan, H. Uys, and M. J. Biercuk, "Arbitrary quantum control of qubits in the presence of universal noise," *New Journal of Physics* **15**, 095004 (2013).

[10] L. Viola and S. Lloyd, "Dynamical suppression of decoherence in two-state quantum systems," *Physical Review A* **58**, 2733 (1998).

[11] E. Knill et al., "Randomized benchmarking of quantum gates," *Physical Review A* **77**, 012307 (2008).

[12] E. Magesan, J. M. Gambetta, and J. Emerson, "Scalable and robust randomized benchmarking of quantum processes," *Physical Review Letters* **109**, 080505 (2012).

[13] E. Magesan, J. M. Gambetta, and J. Emerson, "Characterizing quantum gates via randomized benchmarking," *Physical Review A* **85**, 042311 (2012).

[14] K. Bergmann, H. Theuer, and B. W. Shore, "Coherent population transfer among quantum states of atoms and molecules," *Reviews of Modern Physics* **70**, 1003 (1998).

[15] H. K. Cummins, G. Llewellyn, and J. A. Jones, "Tackling systematic errors in quantum logic gates with composite rotations," *Physical Review A* **67**, 042308 (2003).

[16] M. H. Levitt, "Composite pulses," *Progress in Nuclear Magnetic Resonance Spectroscopy* **18**, 61 (1986).

[17] S. Wimperis, "Broadband, narrowband, and passband composite pulses for use in advanced NMR experiments," *Journal of Magnetic Resonance, Series A* **109**, 221 (1994).

[18] S. L. Braunstein and C. M. Caves, "Statistical distance and the geometry of quantum states," *Physical Review Letters* **72**, 3439 (1994).

[19] T. Caneva, T. Calarco, and S. Montangero, "Chopped random-basis quantum optimization," *Physical Review A* **84**, 022326 (2011).

[20] G. T. Genov, D. Schraft, N. V. Vitanov, and T. Halfmann, "Arbitrarily accurate pulse sequences for robust dynamical decoupling," *Physical Review Letters* **118**, 133202 (2017).

[21] L. M. K. Vandersypen and I. L. Chuang, "NMR techniques for quantum control and computation," *Reviews of Modern Physics* **76**, 1037 (2005).

[22] D. J. Egger and F. K. Wilhelm, "Adaptive hybrid optimal quantum control for imprecisely characterized systems," *Physical Review Letters* **112**, 240503 (2014).

[23] C. Zu, W.-B. Wang, L. He, W.-G. Zhang, C.-Y. Dai, F. Wang, and L.-M. Duan, "Experimental realization of universal geometric quantum gates with solid-state spins," *Nature* **514**, 72 (2014).

[24] J. Kelly et al., "Optimal quantum control using randomized benchmarking," *Physical Review Letters* **112**, 240504 (2014).

A Appendix: QuTiP API Reference

For reference, we summarize key QuTiP 5.x API functions used in this project:

- `qt.basis(N, k)` — Returns $|k\rangle$ in N-dimensional Hilbert space (e.g., `qt.basis(2, 0)` = $|0\rangle$).
- `qt.sigmax()`, `qt.sigmay()`, `qt.sigmaz()` — Pauli matrices as `Qobj`.
- `qt.qeye(N)` — Identity operator on $\mathbb{C}^N$.
- `qt.sesolve(H, psi0, tlist, e_ops=[])` — Schrödinger equation solver. Returns `Result` with `.states` (list of states) and `.expect` (expectation values if `e_ops` provided).
- `qt.expect(op, state)` — Compute $\langle\psi|\hat{O}|\psi\rangle$.
- `qt.fidelity(psi1, psi2)` — State fidelity $|\langle\psi_1|\psi_2\rangle|^2$.
- `qt.Bloch()` — Bloch sphere plotting. Methods: `.add_states(states)`, `.add_points([x, y, z], meth='l')`, `.show()`.

All QuTiP functions preserve quantum object types (`Qobj`) and automatically validate dimensions, Hermiticity, and normalization.