

# Quantum Control Theory and Implementation

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

Rylan Malarchick  
Quantum Controls Simulation Project

October 23, 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

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Motivation and Scope . . . . .	4
1.2	Mathematical Conventions . . . . .	4
<b>2</b>	<b>Phase 1.1: Computational Infrastructure and Reproducibility</b>	<b>4</b>
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
<b>3</b>	<b>Phase 1.2: Drift Hamiltonian and Free Evolution</b>	<b>8</b>
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
<b>4</b>	<b>Phase 1.3: Control Hamiltonian and Pulse Shaping</b>	<b>14</b>
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
<b>5</b>	<b>Phase 2: Optimal Control Theory</b>	<b>19</b>
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
<b>6</b>	<b>Phase 3: Advanced Pulse Shaping and Benchmarking</b>	<b>21</b>
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
<b>7</b>	<b>Future Work</b>	<b>24</b>
7.1	Planned Extensions . . . . .	24
<b>8</b>	<b>Conclusion</b>	<b>24</b>
	<b>References</b>	<b>25</b>
<b>A</b>	<b>Appendix: QuTiP API Reference</b>	<b>25</b>

# 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:

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

- 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  $\geq 3.8$  (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:  $|0\rangle \rightarrow |1\rangle$  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:

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

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

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

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  $2 \times 2$  Hermitian matrices. Any Hermitian operator (including Hamiltonians) can be decomposed as:

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

### 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:

$$\hat{H}_0 = -\vec{\mu} \cdot \vec{B} \quad (5)$$



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

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

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

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

*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:

$$T = \frac{2\pi}{\omega_0} \quad (8)$$

### 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:

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

The formal solution is:

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

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).

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

In matrix form:

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

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

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

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

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

Alternatively, diagonalizing yields (12) directly.  $\square$

### 3.5.2 Evolution of Basis States

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

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

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

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

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

## 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:

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

corresponds to the Bloch vector:

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

The Bloch components are computed from expectation values:

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

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

$$r_z = \langle \psi | \sigma_z | \psi \rangle = |\alpha|^2 - |\beta|^2 \quad (23)$$

### 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  $\omega_0$ . From (18), for initial state  $|\psi(0)\rangle = \alpha |0\rangle + \beta |1\rangle$ :

$$r_x(t) = 2 \operatorname{Re}(\alpha^* \beta e^{-i\omega_0 t}) = r_x(0) \cos(\omega_0 t) + r_y(0) \sin(\omega_0 t) \quad (24)$$

$$r_y(t) = 2 \operatorname{Im}(\alpha^* \beta e^{-i\omega_0 t}) = -r_x(0) \sin(\omega_0 t) + r_y(0) \cos(\omega_0 t) \quad (25)$$

$$r_z(t) = |\alpha|^2 - |\beta|^2 = r_z(0) \quad (26)$$

**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  $\omega_0$ , 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:

```

1 import qutip as qt
2 import numpy as np
3
4 class DriftHamiltonian:
5     """
6     Represents the drift Hamiltonian  $H_0 = -\omega_0/2 * \sigma_z$ 
7     for a two-level quantum system.
8     """
9     def __init__(self, omega_0: float):
10         """
11         Parameters:
12         -----
13         omega_0 : float
14             Qubit transition frequency (Larmor frequency)
15         """
16         self.omega_0 = omega_0
17         self._hamiltonian = -(omega_0 / 2.0) * qt.sigmaz()
18
19     def hamiltonian(self) -> qt.Qobj:
20         """Returns the drift Hamiltonian as a QuTiP Qobj."""
21         return self._hamiltonian
22
23     def eigenvalues(self) -> np.ndarray:
24         """Returns eigenvalues  $[E_0, E_1]$  sorted ascending."""
25         return np.array([-self.omega_0/2, self.omega_0/2])
26
27     def eigenstates(self) -> tuple:
28         """Returns (eigenvalues, eigenstates) as QuTiP arrays."""
29         return self._hamiltonian.eigenstates()
30
31     def period(self) -> float:
32         """Returns the precession period  $T = 2\pi/\omega_0$ ."""
33         return 2 * np.pi / self.omega_0
34
35     def evolve_state(self, psi0: qt.Qobj, t: float) -> qt.Qobj:
36         """
37         Evolves initial state psi0 to time t using analytical
38         propagator.
39         """
40         #  $U(t) = \exp(i * \omega_0 * t / 2 * \sigma_z)$ 
41         propagator = (np.cos(self.omega_0 * t / 2) * qt.qeye(2)
42                     + 1j * np.sin(self.omega_0 * t / 2) * qt.sigmaz()
43                     )
44         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`:

```

1 import qutip as qt

```

```

2 import numpy as np
3
4 class TimeEvolution:
5     """
6     Utilities for time evolution of quantum states under a Hamiltonian.
7     """
8     @staticmethod
9     def evolve_numerical(H: qt.Qobj, psi0: qt.Qobj,
10                        times: np.ndarray) -> qt.solver.Result:
11         """
12         Numerically evolve psi0 under Hamiltonian H using QuTiP's
13         sesolve.
14
15         Returns:
16         -----
17         result : qutip.solver.Result
18             Contains result.states (list of states at each time),
19             result.times (time array).
20         """
21         result = qt.sesolve(H, psi0, times)
22         return result
23
24     @staticmethod
25     def evolve_analytical(H_drift, psi0: qt.Qobj,
26                          times: np.ndarray) -> list:
27         """
28         Analytically evolve psi0 under DriftHamiltonian H_drift.
29
30         Returns:
31         -----
32         states : list of qt.Qobj
33             State at each time in times array.
34         """
35         states = [H_drift.evolve_state(psi0, t) for t in times]
36         return states
37
38     @staticmethod
39     def bloch_coordinates(psi: qt.Qobj) -> tuple:
40         """
41         Compute Bloch sphere coordinates (x, y, z) for state psi.
42         """
43         sx = qt.expect(qt.sigmax(), psi)
44         sy = qt.expect(qt.sigmay(), psi)
45         sz = qt.expect(qt.sigmaz(), psi)
46         return (sx, sy, sz)
47
48     @staticmethod
49     def bloch_trajectory(states: list) -> tuple:
50         """
51         Compute full Bloch trajectory from list of states.
52
53         Returns:
54         -----
55         (x_vals, y_vals, z_vals) : tuple of np.ndarray
56         """
57         coords = [TimeEvolution.bloch_coordinates(psi)
58                   for psi in states]
59         x_vals = np.array([c[0] for c in coords])

```

```

59     y_vals = np.array([c[1] for c in coords])
60     z_vals = np.array([c[2] for c in coords])
61     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$	$2.3 \times 10^{-15}$	$1.1 \times 10^{-15}$
$ 1\rangle$	$1.8 \times 10^{-15}$	$9.7 \times 10^{-16}$
$ +\rangle = ( 0\rangle +  1\rangle)/\sqrt{2}$	$3.1 \times 10^{-15}$	$1.4 \times 10^{-15}$
$ -\rangle = ( 0\rangle -  1\rangle)/\sqrt{2}$	$2.9 \times 10^{-15}$	$1.3 \times 10^{-15}$
$ i+\rangle = ( 0\rangle + i 1\rangle)/\sqrt{2}$	$3.4 \times 10^{-15}$	$1.5 \times 10^{-15}$

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:

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

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  $\omega_d$  (drive frequency) couples to the qubit's dipole moment. The interaction Hamiltonian is:

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

where:

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

- $\omega_d$  is the drive frequency
- $\phi$  is the pulse phase
- $\sigma_x$  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  $\omega_d$ . Define the unitary transformation:

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

In the rotating frame, the Hamiltonian becomes:

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

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

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

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

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

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

### 4.4 Rabi Oscillations

#### 4.4.1 Constant Driving

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

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

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

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

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

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

**Theorem 4.1** (Rabi Oscillations). Under constant on-resonance driving at Rabi frequency  $\Omega_0$ , 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:

- **$\pi$ -pulse (X-gate):** Duration  $T_\pi = \pi/\Omega_0$  achieves  $|0\rangle \rightarrow |1\rangle$
- **$\pi/2$ -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  $\theta$  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:

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

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

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

### 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:

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

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

The control Hamiltonian becomes:

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

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

## 4.6 Detuning Effects

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

$$\Omega_{\text{eff}} = \sqrt{\Omega^2 + \Delta^2} \quad (41)$$

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

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

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`:



```

1 class ControlHamiltonian:
2     def __init__(self, pulse_func, drive_axis='x',
3                   phase=0.0, detuning=0.0):
4         # Store pulse envelope function Omega(t)
5         # Configure drive axis (x, y, or xy for DRAG)
6         # Set phase and detuning parameters
7
8     def hamiltonian(self, t):
9         # Return H_c(t) at time t
10        # Apply RWA with phase rotation
11
12    def evolve_state(self, psi0, times, H_drift=None):
13        # Evolve state under H_total = H_drift + H_c(t)
14        # Use QuTiP's sesolve with time-dependent H
15
16    def gate_fidelity(self, psi0, psi_target, times):
17        # 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:  $\pi$ -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
$\pi$ -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
- $\pi$  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.  $\Delta$
- 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:  $\pi$  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  $U_{\text{target}}$ , 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:

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

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

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

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

#### 5.2.2 Constrained Optimization

The optimal control problem is formulated as:

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

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:

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

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

Krotov's method provides monotonic convergence guarantees through a penalty-based update:

$$u_j^{(n+1)}(t) = u_j^{(n)}(t) + \frac{1}{\lambda} \operatorname{Im} \left[ \left\langle \chi^{(n)}(t) \left| H_j \right| \psi^{(n)}(t) \right\rangle \right] \quad (47)$$

where  $\lambda > 0$  is a step-size parameter controlling convergence rate.

### Advantages over GRAPE:

- Monotonic convergence:  $F^{(n+1)} \geq F^{(n)}$
- No line search required
- Smooth pulse updates

## 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:**  $t \rightarrow t + \delta t$

The sensitivity is quantified by:

$$S_\theta = \left| \frac{\partial F}{\partial \theta} \right| \quad (48)$$

for parameter  $\theta$  (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:

$$\langle F \rangle = \int F(\Omega + \delta\Omega) P(\delta\Omega) d\delta\Omega \quad (49)$$

## 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:

$$H_{\text{DRAG}}(t) = \Omega(t) \cos(\omega t) \sigma_x - \beta \dot{\Omega}(t) \sin(\omega t) \sigma_y \quad (50)$$

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

**Theoretical Basis:** The  $\sigma_y$  term cancels the leading-order leakage error:

$$\epsilon_{\text{leakage}} \propto \frac{\Omega^2}{\delta^2} \rightarrow 0 \text{ with DRAG} \quad (51)$$

#### 6.2.2 Composite Pulses

Composite pulses achieve robustness through sequences of imperfect operations that cancel systematic errors. For example, the BB1 (Broadband 1) sequence:

$$U_{\text{BB1}} = X_{\phi_3} X_{\phi_2} X_{\phi_1} X_{\phi_0} \quad (52)$$

with phases  $\phi = [\pi, 3\pi, 3\pi, \pi]$  compensates for pulse amplitude errors to second order.

**Key Composite Sequences:**

- **BB1:** Broadband, compensates amplitude and detuning errors
- **CORPSE:** Compensation for Off-Resonance with Pulse Sequence
- **SCROFULOUS:** Short Composite Rotation For Undoing Length Over- and Under-Shoot

#### 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:

$$\left| \frac{\langle m(t) | \dot{H}(t) | n(t) \rangle}{(E_n - E_m)^2} \right| \ll 1 \quad (53)$$

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

$$H_{\text{STIRAP}}(t) = \Omega_p(t) |1\rangle \langle 0| + \Omega_s(t) |2\rangle \langle 1| + \text{h.c.} \quad (54)$$

with Stokes pulse  $\Omega_s$  before pump  $\Omega_p$ .

### 6.3 Task 2: Gate Library and Compilation

#### 6.3.1 Universal Single-Qubit Gates

Any single-qubit unitary can be decomposed as:

$$U = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta) \quad (55)$$

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:

$$U = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix} = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta) \quad (56)$$

with:

$$\gamma = 2 \arccos(|u_{00}|) \quad (57)$$

$$\beta = \arg(u_{00}) - \arg(u_{01}) \quad (58)$$

$$\delta = \arg(u_{00}) + \arg(u_{01}) \quad (59)$$

#### 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

Filter functions characterize pulse sensitivity to noise at different frequencies:

$$F(\omega) = \left| \int_0^T y(t) e^{i\omega t} dt \right|^2 \quad (60)$$

where  $y(t)$  is the control modulation.

**Noise Infidelity:**

$$\chi = \frac{1}{2\pi} \int F(\omega) S(\omega) d\omega \quad (61)$$

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

**Common Noise Models:**

- White noise:  $S(\omega) = S_0$
- $1/f$  noise:  $S(\omega) = A/|\omega|^\alpha$
- Lorentzian:  $S(\omega) = A/(1 + (\omega/\omega_c)^2)$

#### 6.4.2 Randomized Benchmarking

Randomized Benchmarking (RB) extracts average gate fidelity via decay of survival probability:

$$F_{\text{seq}}(m) = Ap^m + B \quad (62)$$

for sequence length  $m$ , with depolarizing parameter  $p$  related to average gate fidelity:

$$F_{\text{avg}} = 1 - \frac{(1-p)(d-1)}{d} \quad (63)$$

**Clifford Group:** The 24-element single-qubit Clifford group is generated by Hadamard  $H$  and phase gate  $S = \text{diag}(1, i)$ .

**Interleaved RB:** Characterizes specific gate  $G$  by interleaving it between random Cliffords:

$$F_G = 1 - \frac{(1 - p_{\text{int}}/p_{\text{std}})(d-1)}{d} \quad (64)$$

#### 6.4.3 Fisher Information

Quantum Fisher information quantifies parameter estimation precision:

$$F(\theta) = \text{Tr}[\rho L^2] \quad (65)$$

where  $L$  is the symmetric logarithmic derivative satisfying:

$$\frac{\partial \rho}{\partial \theta} = \frac{1}{2}(L\rho + \rho L) \quad (66)$$

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

$$\text{Var}(\hat{\theta}) \geq \frac{1}{NF(\theta)} \quad (67)$$

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] 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).
- [6] 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).
- [7] 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).
- [8] E. Knill et al., “Randomized benchmarking of quantum gates,” *Physical Review A* **77**, 012307 (2008).
- [9] E. Magesan, J. M. Gambetta, and J. Emerson, “Scalable and robust randomized benchmarking of quantum processes,” *Physical Review Letters* **109**, 080505 (2012).
- [10] 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).
- [11] 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).
- [12] S. L. Braunstein and C. M. Caves, “Statistical distance and the geometry of quantum states,” *Physical Review Letters* **72**, 3439 (1994).

## 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.