Controlling a Superconducting Quantum Computer

Daniel Cohen Hillel

Supervisor: Dr. Serge Rosenblum

Advanced Project in Physics A (20382)

The Open University of Israel

The Weizmann Institute

Contents

Abstract

(?)

Quantum computing presents many challenges, we're going to try to tackle one of them, Quantum Optimal Control. How do we control the quantum computer? What do you need to send to make what you want happen? How do you physically build control the transmission of the control pulses and what the pulses even look like? These are all question we are going to try to answer throughout this project.

The project starts with an introduction to quantum computing. This is the basic of quantum computing and is only meant for readers who have no knowledge of quantum computing. Concepts such as the qubit and operations are introduced there.

In the section chapter we define the system and characterize it. We use quantum optics and the Jaynes-Cumming model to characterize the system. This is where the bulk of the pure physics is done.

The third chapter is the main interest of this project, quantum optimal control and the GRAPE algorithm. In this chapter we show how can you use the system characterization we made in chapter 2 to find the pulses you need to send to control the quantum computer as you want.

The fourth and last chapter talks about the physical implementation of the transmission of the pulses we found in the previous chapter. We show how the entire system is connected, from the AWG[1] to the actual qubits. We discuss the challenges of creating the pulses and how to solve these problems.

[1]Arbitrary Waveform Generator, read the chapter to learn more

1 Introduction

1.1 What is a Quantum Computer?

We'll assume the reader is familiar with the basics of computing and quantum mechanics, here's a brief overview.
A classical computer is, essentially, a calculator, not of "Regular" numbers but of *binary numbers*. A *binary digit*("*bits*" from now on) can be in one of two states, usually represented by 0 and 1. We can use *logic gates* to control and manipulate bits to do all kinds of calculations. These are the building blocks of the classical computer, with the ability to do calculation with bits, and the ability to store bits in the memory we are able to construct a computer.

So what is a quantum computer then? Well, if the classical computer uses bits to do calculations, a quantum computer uses *quantum bits*("*qubits*" from now on) for it's calculations. A qubit, much the same as a bit, has 2 states, a 0 state and a 1 state(notated $|0\rangle$ and $|1\rangle$ for reasons we'll see later), the difference is that a qubit can be in a *superposition* of the 2 states, we can use this property to our advantage to create new type of computation.

1.2 Qubits and Quantum Gates

Mathematically, we think of qubits as 2-dimensional vectors, where the first term corresponds to the $|0\rangle$ state and the second term corresponds to the $|1\rangle$ state, so a qubit in a state $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ can be represented as

$$\begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

The value of each state is correlated to the probability of the qubit to be in that state. The probability is given by the absolute value of that state squared, so in this example, the qubit has a 50% chance to be in the $|0\rangle$ state and a 50% chance to be in the $|1\rangle$ state.

In this world of qubits as vectors, we think of logic gates, known as *quantum gates*, as

matrices[2], and when the qubit goes through the logic gate, the result is multiplying the matrix by the qubit. Let's see an example for one of the simplest logic gates we have in classical computing, the NOT gate, a quantum implementation of this gate takes $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$), the matrix that achieves this is

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Known as σ_1(Pauli matrix 1). Let's look at a simple example to see how this works, if we input $|0\rangle$ into the NOT quantum gate, we get as a result

$$NOT\,|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$$

As we expected, NOT $|0\rangle$ is $|1\rangle$. There are many more 1-qubit quantum gates we can think of(infinite amount actually). The interesting thing is that in classical computing, we only have 4 possible logic gates on a single bit(compared to infinite amount for qubit), these are the identity(do nothing) gate, the NOT gate, the output always 1 gate, and output always 0 gate. There are quantum equivalent to all of these gates, we won't get into that now.

The last thing we need to know to understand the basic of quantum computing, is how to treat multiple qubits. If we have several of qubits in our system, we think of all the qubits together as one vector that is the *tensor product*[3] of all the qubits. Lets say we have a $|0\rangle$ and $|1\rangle$ qubits in our system, we represent that by $|01\rangle$ and it is equal to

$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

And a quantum gate on multiple qubits is simply a larger matrix.

The thing about the tensor product is that for N qubits, it has 2^N coefficients! This is yet

[2]Under the constraint that the matrices are unitary

[3]Represented as a Kronecker product for the dimensions sake

another clue of the power that quantum computers have compared to classical computers.

Now that we have the basic tools of quantum computing, we can use them to get motivation for the amazing things quantum computers can do

1.3 Algorithms and Further motivation

> "Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy."
>
> - Richard Feynman

Quantum computation is a cool idea and all, but it becomes much more interesting when you think of the possibilities of its uses. From simulation of drugs for developments of new cures to unbreakable encryption, quantum computing promises a lot.

One of the most famous algorithms in quantum computing is *Shor's algorithm*. Shor's algorithm is a quantum algorithm to factor large numbers, in classical computing the best way to factor a number is to try all the numbers smaller then it and check if they devise the number. This is a really slow way to do it and basically impossible for computers to do(it might take thousands of years to factor one number on a classical computer). For this reason, almost everything is encrypted in a way that to decrypt information you're not supposed to see, you need to factor a really big number[4]. With classical computers this is a nearly impossible problem that could take thousands of years, but quantum computers can solve it in seconds with Shor's algorithm. This is a great example of the power in quantum computing.

1.4 Superconducting Quantum Computers

The physical implementation of the qubit itself isn't to subject of this project but we can look a bit on how you would implement such a thing. A problem we face when making a quantum computer is would physical phenomena would actually be the qubit. For classical computer we already have this figured out for years, the bit is the voltage on a wire, 1 is

[4]This is called RSA encryption

one there is voltage on the wire and 0 is if there's none, simple. For a quantum computer this is much more complicated, there are many quantum phenomena we can use as our qubit, such as the energy level of an atom, the spin of an electron, the polarization of photons and so on. This project is about a *superconducting* quantum computer, with superconducting qubits.

Superconducting qubits are microwave circuits in which the cooper-pair condensate effectively behaves as a one-dimensional quantized particle. By inserting Josephson junctions, the circuit can be made nonlinear, allowing us to isolate the qubit from higher-energy levels and treat it as a two level system(instead of the many level system that it really is).

2 Quantum Optics

In this chapter we'll introduce concepts in *Quantum Optics*. It's important to have at least a little understanding of quantum optics before reading the project. The implementation of quantum computers that we discuss in this project is one where the qubit is some sort of two level system(like an atom with two levels) interacting with a cavity. The cavity is some box made out of a conducting material. In the cavity the quantization of light could occur and we'll learn more about this phenomena in the next sections.

This won't be an in depth review of quantum optics and wouldn't even scratch the surface of the subject. If you want a much better introduction the the subject I highly recommend "Introductory Quantum Optics" by Christopher C. Gerry and Peter L. Knight. This chapter will only touch the subjects related and used by the project.

2.1 The Quantization of the Electromagnetic Field

2.1.1 The Homogeneous Electromagnetic Equation

We'll start from Maxwell's equations, proving one of the most important result of the electromagnetic theory that you've properly encountered before in you classical electromagnetism course.

Maxwell's equations in free space are:

$$\boldsymbol{\nabla} \cdot \mathbf{E} = 0 \tag{2.1a}$$

$$\boldsymbol{\nabla} \cdot \mathbf{B} = 0 \tag{2.1b}$$

$$\boldsymbol{\nabla} \times \mathbf{E} = \frac{\partial \mathbf{B}}{\partial t} \tag{2.1c}$$

$$\boldsymbol{\nabla} \times \mathbf{B} = \mu_0 \epsilon_0 \frac{\partial \mathbf{E}}{\partial t} \tag{2.1d}$$

Taking the curl of 2.1c and 2.1d we get

$$\boldsymbol{\nabla} \times (\boldsymbol{\nabla} \times \mathbf{E}) = \boldsymbol{\nabla} \times (-\frac{\partial \mathbf{B}}{\partial t}) = -\frac{\partial}{\partial t}(\boldsymbol{\nabla} \times \mathbf{B}) = -\mu_0 \epsilon_0 \frac{\partial^2 E}{\partial t^2} \tag{2.2a}$$

$$\boldsymbol{\nabla} \times (\boldsymbol{\nabla} \times \mathbf{B}) = \boldsymbol{\nabla} \times (\mu_0 \epsilon_0 \frac{\partial E}{\partial t}) = \mu_0 \epsilon_0 \frac{\partial}{\partial t}(\boldsymbol{\nabla} \times E) = -\mu_0 \epsilon_0 \frac{\partial^2 \mathbf{B}}{\partial t^2} \tag{2.2b}$$

We can use the vector identity

$$\nabla \times (\nabla \times \mathbf{V}) = \nabla (\nabla \cdot \mathbf{V}) - \nabla^2 \mathbf{V} \tag{2.3}$$

And obtain from 2.2a and 2.2b

$$\nabla(\nabla \cdot \mathbf{E}) - \nabla^2 \mathbf{E} = -\mu_0 \epsilon_0 \frac{\partial^2 \mathbf{E}}{\partial t^2} \tag{2.4a}$$

$$\nabla(\nabla \cdot \mathbf{B}) - \nabla^2 \mathbf{B} = -\mu_0 \epsilon_0 \frac{\partial^2 \mathbf{B}}{\partial t^2} \tag{2.4b}$$

Now, we can use 2.1a and 2.1b To cancel the left most term and get

$$\nabla^2 \mathbf{E} = \mu_0 \epsilon_0 \frac{\partial^2 \mathbf{E}}{\partial t^2} \tag{2.5a}$$

$$\nabla^2 \mathbf{B} = \mu_0 \epsilon_0 \frac{\partial^2 \mathbf{B}}{\partial t^2} \tag{2.5b}$$

Now because we know that $v_{ph} = \frac{1}{\sqrt{\mu_0 \epsilon_0}}$ and that the phase velocity of electromagnetic waves in a vacuum is the speed of light, c_0, we get

$$\begin{aligned} \nabla^2 \mathbf{E} &= \frac{1}{c_0^2} \frac{\partial^2 \mathbf{E}}{\partial t^2} \\ \nabla^2 \mathbf{B} &= \frac{1}{c_0^2} \frac{\partial^2 \mathbf{B}}{\partial t^2} \end{aligned} \tag{2.6}$$

These equation are called *the homogeneous electromagnetic wave equations*. We'll pick a polarization arbitrarily to be in the x direction(that way we get only the component of the electric field and the y component of the magnetic field, E_x and B_y) so now we get,

$$\begin{aligned} \frac{\partial^2 E_x}{\partial x^2} &= \frac{1}{c_0^2} \frac{\partial^2 E_x}{\partial t^2} \\ \frac{\partial^2 B_y}{\partial y^2} &= \frac{1}{c_0^2} \frac{\partial^2 B_y}{\partial t^2} \end{aligned} \tag{2.7}$$

2.1.2 The Single Mode Cavity

Now that we have the homogeneous electromagnetic field equations at hand we can solve them to get the quantization of light. We can solve 2.7 using separation of variables,

$$E_x(z,t) = Z(z)T(t)$$

Yielding the solution,

$$\begin{aligned} E_x(z,t) &= \sqrt{\frac{2\omega_c^2}{V\epsilon_0}} q(t) \sin kz \\ B_y(z,t) &= \sqrt{\frac{2\mu_0}{V}} \dot{q}(t) \cos kz \end{aligned} \tag{2.8}$$

where V is the effective volume of the cavity, q is a time-dependent amplitude with units of length, and $k = m\pi/L$ for an integer $m > 0$

The Hamiltonian is given by

$$H = \frac{1}{2} \int \epsilon_0 \mathbf{E}^2 + \frac{\mathbf{B}^2}{\mu_0} dV \tag{2.9}$$

$$= \frac{1}{2} \int \epsilon_0 E_x^2(z,t) + \frac{B_y^2(z,t)}{\mu_0} dz \tag{2.10}$$

$$= \frac{1}{2}[\dot{q}^2(t) + \omega_c^2 q^2(t)] \tag{2.11}$$

Now, going from dynamical variables to operators(considering $\dot{q} \equiv p$) we get,

$$\hat{E}_x(z,t) = \sqrt{\frac{2\omega_c^2}{V\epsilon_0}} \hat{q}(t) \sin kz \tag{2.12}$$

$$\hat{B}_y(z,t) = \sqrt{\frac{2\mu_0}{V}} \hat{p}(t) \cos kz \tag{2.13}$$

$$\hat{H} = \frac{1}{2}[\hat{p}^2(t) + \omega_c^2 \hat{q}^2(t)] \tag{2.14}$$

This is the same Hamiltonian as for the harmonic oscillator. **A single mode cavity acts like a harmonic oscillator**.

Let's introduce creation and annihilation operators,

$$\hat{a}(t) = \frac{1}{\sqrt{2\hbar\omega_c}}[\omega_c\hat{q}(t) + i\hat{p}(t)]$$

$$\hat{a}^\dagger(t) = \frac{1}{\sqrt{2\hbar\omega_c}}[\omega_c\hat{q}(t) - i\hat{p}(t)]$$

We can write the electric and magnetic field as,

$$\hat{E}_x(z,t) = E_0[\hat{a}(t) + \hat{a}^\dagger(t)]\sin kz \tag{2.15}$$

$$\hat{B}_y(z,t) = \frac{E_0}{c}[\hat{a}(t) - \hat{a}^\dagger(t)]\cos kz \tag{2.16}$$

And we can write the Hamiltonian as,

$$\hat{H}_{cavity} = \hbar\omega_c[\hat{a}\hat{a}^\dagger + \frac{1}{2}] \approx \hbar\omega_c\hat{a}\hat{a}^\dagger \tag{2.17}$$

We can ignore the zero-point energy $\frac{\hbar\omega_c}{2}$ if we define it as the zero energy point.

We'll now do a big jump to the Jaynes-Cummings model, usally it comes much later when studying quantum optics but since this is only a small chapter and the Jayness-Cumming model is the model we're going to use to describe the system, we'll look at it now.

2.2 The Jaynes–Cummings Model

Our goal is to mathematically model the Hamiltonian of a system of a two-level atom interacting with a single quantized mode of an optical cavity's electromagnetic field.

First we'll divide the system into 3 parts, The atom(it can be other two-level quantum systems), the cavity(electromagnetic field with quantized modes) and the interaction between the atom and the cavity(an atom can emit a photon to the cavity and change it's electromagnetic field, or catch a photon from the cavity and go up an energy level).

Let's start with the cavity(we'll consider a one dimensional cavity for now).

2.2.1 The Hamiltonians

We want to separate the total Hamiltonian into approachable parts, we can separate it like so,

$$H = H_{atom} + H_{cavity} + H_{interaction}$$

where the atom and cavity Hamiltonians are the Hamiltonian of the atom and cavity if they were the only part of the (closed)system and the interaction Hamiltonian is the Hamiltonian of the interaction between the atom and the cavity in the system.

cavity

We already calculated the Hamiltonian of the cavity and it is given by equation 2.17 as

$$\boxed{\hat{H}_{cavity} = \hbar\omega_c \hat{a}\hat{a}^\dagger} \tag{2.18}$$

For completeness sake I've included this paragraph here even though we calculated the Hamiltonian earlier.

atom

Now that we have the cavity's Hamiltonian, we can go on to calculate the atom(qubit) Hamiltonian.
Remember that the qubit is a 2-level system, meaning we can define it has a superposition of the ground, $|g\rangle$, and excited, $|e\rangle$, states. The energy of the atom is the sum of the energy of each state times it's energy($\sum E_s P(|s\rangle)$). The probability to be in a state $|s\rangle$ is given by $|s\rangle\langle s|$ so we can write,

$$\hat{H}_{atom} = E_g |g\rangle\langle g| + E_e |e\rangle\langle e| \tag{2.19}$$

Using the vector representation of these states we'll write,

$$\begin{aligned}\hat{H}_{atom} &= E_e \begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix} + E_g \begin{bmatrix}0 & 0\\ 0 & 1\end{bmatrix} = \begin{bmatrix}E_e & 0\\ 0 & E_g\end{bmatrix} \\ &= \frac{1}{2}\begin{bmatrix}E_g + E_e & 0\\ 0 & E_g + E_e\end{bmatrix} + \frac{1}{2}\begin{bmatrix}E_e - E_g & 0\\ 0 & -(E_e - E_g)\end{bmatrix} \\ &= \frac{1}{2}(E_g + E_e)\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} + \frac{1}{2}(E_e - E_g)\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} \\ &= \frac{1}{2}(E_g + E_e)\mathbb{I} + \frac{1}{2}(E_e - E_g)\hat{\sigma}_z\end{aligned}$$

Again, we can define the zero point energy so that the first term becomes 0. We know the difference between the excited state energy and the ground state energy because it's approximately an harmonic oscillator so $E_e - E_g = \hbar\omega_a$ where ω_a is the atom frequency. Now we can write,

$$\boxed{\hat{H}_{atom} = \frac{1}{2}\hbar\omega_a\hat{\sigma}_z} \tag{2.20}$$

interaction

Now for the last part of the Hamiltonian, we want to know the interaction Hamiltonian between the atom and the cavity. The interaction Hamiltonian is now

$$\hat{H}_{interaction} = -\hat{\mathbf{d}}\cdot\hat{\mathbf{E}} = -\hat{d}E_0 \sin kz(\hat{a} + \hat{a}^\dagger)$$

Where we introduced $\hat{d} = \hat{\mathbf{d}}\cdot\hat{x}$ (remember that we defined the axis so that x is the direction of polarization of the electromagnetic field). We'll also introduce the atomic transition operators

$$\hat{\sigma}_+ = |e\rangle\langle g|, \qquad \hat{\sigma}_- = |g\rangle\langle e| = \hat{\sigma}_+^\dagger$$

and the inversion operator

$$\hat{\sigma}_3 = |e\rangle\langle e| - |g\rangle\langle g|$$

only the off-diagonal elements of the dipole operator are nonzero so we may write

$$\hat{d} = d|e\rangle\langle g| + d^*|g\rangle\langle e| = d\hat{\sigma}_- + d^*\hat{\sigma}_+ = d(\hat{\sigma}_+ + \hat{\sigma}_-)$$

thus the interaction Hamiltonian is

$$\hat{H}_{interaction} = \hbar\lambda(\hat{\sigma}_+ + \hat{\sigma}_-)(\hat{a} + \hat{a}^\dagger) = \hbar\lambda(\hat{\sigma}_+\hat{a} + \hat{\sigma}_+\hat{a}^\dagger + \hat{\sigma}_-\hat{a} + \hat{\sigma}_-\hat{a}^\dagger) \tag{2.21}$$

We shown the the operators $\hat{a}$ and $\hat{a}^\dagger$ evolve as

$$\hat{a}(t) = \hat{a}(0)e^{-i\omega t}, \qquad \hat{a}^\dagger(t) = \hat{a}^\dagger(0)e^{i\omega t} \tag{2.22}$$

And similarly we can show for the free-atomic case that

$$\hat{\sigma}_\pm(t) = \hat{\sigma}_\pm(0)e^{\pm i\omega t} \tag{2.23}$$

We can write while approximating $\omega_0 \approx \omega$

$$\begin{aligned} \hat{\sigma}_+\hat{a} &\sim e^{i(\omega_0-\omega)t} \\ \hat{\sigma}_-\hat{a}^\dagger &\sim e^{-i(\omega_0-\omega)t} \\ \hat{\sigma}_+\hat{a}^\dagger &\sim e^{i(\omega_0+\omega)t} \\ \hat{\sigma}_-\hat{a} &\sim e^{-i(\omega_0+\omega)t} \end{aligned} \tag{2.24}$$

We can see that the last two term vary much more rapidly than the first two. Furthermore, the last two terms do not conserve energy(They correlate to [photon addition + atom excitation] and [photon reduction + atom grounded]), we're going to drop the last fast rotating terms[5] and finally get

$$\boxed{H_{interaction} = \hbar\lambda(\hat{\sigma}_+\hat{a} + \hat{\sigma}_-\hat{a}^\dagger)} \tag{2.25}$$

Finally, we can write the full JC Hamiltonian

$$\boxed{\hat{H} = \frac{1}{2}\hbar\omega_0\hat{\sigma}_3 + \hbar\omega\hat{a}^\dagger\hat{a} + \hbar\lambda(\hat{\sigma}_+\hat{a} + \hat{\sigma}_-\hat{a}^\dagger)} \tag{2.26}$$

[5]This is called the *Rotating Wave Approximation* or RWA in short

2.2.2 Effective Hamiltonian at the Disspersive Limit

...

2.2.3 Interaction with the Classical Field

As you might have noticed, we are classical creatures, I'm (unfortunately) not in a superposition of being here and on the moon at the same time :(. As classical creatures, if we want to interact with the quantum world we need to do so with in classical means with a classical interface to the quantum world. Back to the Jaynes-Cummings model, what happens when we introduce a classical electromagnetic field(such as the drives of the system, which are the main subject of this project)

2.3 Coherent States

...

2.4 Fock States(Number States)

...

3 Optimal Control

This chapter is the main topic of this project, we tackle the problem of *Quantum Optimal Control*. If we have a quantum system, the way we control the system is by sending some (classical)electromagnetic pulse to the system. Now an obvious question arises, what is this pulse? How does it look like? sin? cos? In what frequency? Some other wave? What does it do?

Turns out this is not that simple of a question. In the rest of the chapter we'll try to give an answer to this question using something called *GRAPE*.

3.1 What's GRAPE

Although in some cases we can calculate the pulse analytically, most of the time this isn't an option, so we need to use numeric means to find the pulse. To find the desired pulse through numeric means, we can model our system on a normal(boring) computer and simulate what happens when you send a pulse, then, we can try to change the pulse(in a smart way) until we get the desired effect. So for example, let's say we want to find the wave pulse that corresponds to the NOT gate, we can start by guessing some random wave(constant zero, sin and so on), the random wave probably won't act as a NOT gate, then, we change the wave a little bit many times and on each iteration the pulse acts more and more as a NOT gate. So what's GRAPE than? The ***GR****adient* ***A****scent* ***P****ulse* ***E****ngineering* was first proposed in [2]. When we model our system, we treat the wave as a step-wise constant function, so the wave is just an array with many variables and we want to find the best values that give the result that we want. then we set a cost function [6] that tells us how are the values of the wave to give the wanted result. This cost function is a many dimensional function(each step of the wave is a dimension of the cost function), and we can find its gradient. Using the cost function and it's gradient we can use some optimization algorithm(mainly, the L-BFGS-B method) to find the maximum of the cost function that gives us the optimal wave to send to the cavity.

[6] discussed in details in the next section

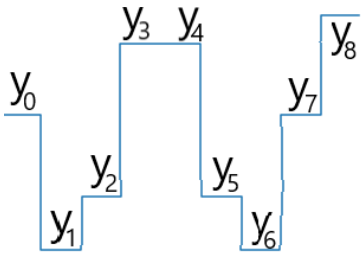


Figure 3.1: Example of a step-wise constant function as an array of numbers

3.2 The Cost Function

So what is this cost function really? ***Fidelity***. It measures the "closeness" of two quantum states and varies between 0 and 1. So for example, the fidelity between state $|0\rangle$ and state $|1\rangle$ is equal to 0, because they are the most different two quantum states can be, while for example the fidelity between state $|0\rangle$ and state $|0\rangle$ is equal to 1, because they are the closest two states can be to each other(for a matter of fact, every state has fidelity 1 with itself and end fidelity 0 with an opposite state). But still, how do we calculate the fidelity between two states? well, turns out its very simple, it's just their product[7].

$$F(\psi_1, \psi_2) = |\langle\psi_1|\psi_2\rangle|^2 \tag{3.1}$$

We want to maximize the fidelity with GRAPE. In a previous chapter we characterized the Hamiltonian of the system(equation (num)), so now we can use the good old *time-dependent Schrodinger equation*:

$$i\hbar\frac{d}{dt}|\Psi(t)\rangle = \hat{H}|\Psi(t)\rangle \tag{3.2}$$

The hamiltonian of the system, as given in section 2.4, is of the form

$$H(t) = H_0 + \sum_k \epsilon_k(t) H_k \tag{3.3}$$

where H_0 is the hamiltonian of the system without the drives(given, for example, from the Jaynes-Cummings model or some other description of the system). Each $\epsilon_k(t)$ is the

[7]Assuming both states are pure states and not density matrices

amplitude as a function of time of the control drive pulse, and each H_k is the hamiltonian describing the control pulse interaction with the system, we call these hamiltonian the *drive hamiltonians*. Our goal with GRAPE is to find $\epsilon_k(t)$.

Because of the way this Hamiltonian is built, on each constant step of the wave function, the entire Hamiltonian is constant, and luckily for us, the solution of the Schrodinger equation for a constant Hamiltonian is pretty simple and given by

$$U(t) = e^{-\frac{i}{\hbar}\int_{T_0}^{T_1} H(t)dt} \tag{3.4}$$

Because we chose T_0 and T_1 as the end points of a step of the functions, the total Hamiltonian of the system is constant so the integral is just a simple multiplication by $T_1 - T_0$ which we'll write as δt. So the solution is

$$U(t) = e^{-\frac{i\cdot\delta t}{\hbar}H(t)} \tag{3.5}$$

In order to calculate the solution over all the pulse we need to solve for the first step, find the solution by the end of the time-step, then use it as the initial condition for the next time step. If we do so, we get that after N time steps, the solution over these time step is simply the product of the solutions until that time step

$$U(\epsilon(t)) = \prod_{k=1}^{N} U_k \tag{3.6}$$

Now with $U(\epsilon_N)$ in our hands, we can calculate the evolution of the state over time

$$|\Psi_{final}\rangle = U(\epsilon_N)\,|\Psi_{initial}\rangle \tag{3.7}$$

This way if we want to calculate the fidelity after applying the drives we can simply calculate the fidelity between the wanted state and the final state,

$$F(\vec{\epsilon(t)}) = F(\Psi_{target}, \Psi_{final}) = |\langle\Psi_{target}|\,U_N\,|\Psi_{initial}\rangle|^2 \tag{3.8}$$

Now, theoretically we can use an algorithm to try different waves until we find a wave that does what we want(brute force for example), but this will take to much time and the

computation won't finish in any reasonable amount of time. Because of this, we'll want to use a smart search algorithm(such as L-BFGS-B) but to do so we need the gradient of the cost function(the variables of the cost function are the values of the steps of the drive pulses).

3.3 The Gradient

As mentioned, if we want to optimize the cost function efficiently we'll need to calculate the gradient of the cost function and use an optimization algorithm. We can obviously use the finite difference method to calculate the gradient but this method is heavy on the computation and using it kind of defeats the purpose of the optimization algorithm to be more efficient. We'll take a smarter approach to calculating the gradient.

We can start with looking at the expression for the overlap of the final state and the target wanted state. This is very similar to the fidelity, if we take the absolute value of the overlap and square it we get the fidelity. We'll work with the overlap for now since it's nicer to work with because it does not have an absolute value in the calculation, we'll go from the overlap back to the fidelity in the end.

$$c = \langle \Psi_{target} | \Psi_{final} \rangle = \langle \Psi_{target} | \, U \, | \Psi_{initial} \rangle \tag{3.9}$$

We want to differentiate this expression by each control parameter. U is defined as:

$$U = U_N U_{n-1} ... U_2 U_1$$

And when differentiating by a control parameter only one U_k is affected, so we can write,

$$\frac{\partial c}{\partial \epsilon_k} = \langle \Psi_{target} | \, U_N U_{N-1} ... U_{k+1} \frac{\partial U}{\partial \epsilon_k} U_{k-1} ... U_2 U_1 \, | \Psi_{initial} \rangle$$

We can write that for a constant Hamiltonian(from Schrodinger's equation)

$$U_k = e^{-\frac{i \cdot \delta t}{\hbar} H(t)}$$

We can approximate the derivative $\frac{\partial U_k}{\partial \epsilon_k}$ in the limit of small δt by writing

$$\frac{\partial U_k}{\partial \epsilon_k} \approx -\frac{i \cdot \delta t}{\hbar} \frac{\partial H}{\partial \epsilon_k} \cdot e^{-\frac{i \cdot \delta t}{\hbar} H(t)} = -\frac{i \cdot \delta t}{\hbar} \frac{\partial H}{\partial \epsilon_k} U_k$$

We can use this expression as the gradient values but it's still rather complex computationally($o(N^2)$ complexity).

We can use a bit different method to calculate the gradient to save on the computation by reducing the overhead.

Now the derivative of the cost function by an amplitude of the wave has become

$$\frac{dc}{d\epsilon_k} = -\frac{i \cdot \delta t}{\hbar} \langle \Psi_{target} | U_N U_{N-1} ... U_{k+1} \frac{dH}{d\epsilon_k} U_k ... U_2 U_1 | \Psi_{initial} \rangle \tag{3.10}$$

Let's define 2 wave wave functions ψ_{bwd} and ψ_{fwd}, they will be the multiplication component before and after the derivative of H, so:

$$\frac{\partial c}{\partial \epsilon_k} = -\frac{i \cdot \delta t}{\hbar} \left\langle \psi_{bwd}^{(k+1)} \right| \frac{\partial H}{\partial \epsilon_k} \left| \psi_{fwd}^{(k)} \right\rangle \tag{3.11}$$

We can easily see from 3.10 that

$$\left| \psi_{fwd}^{(k)} \right\rangle = \begin{cases} |\psi_{init}\rangle & k = 0 \\ U_k \left| \psi_{fwd}^{(k-1)} \right\rangle & otherwise \end{cases}$$

$$\left| \psi_{bwd}^{(k)} \right\rangle = \begin{cases} |\psi_{targ}\rangle & k = N+1 \\ U_k^\dagger \left| \psi_{bwd}^{(k+1)} \right\rangle & otherwise \end{cases}$$

Now all we need is to do $2N$ calculations in the beginning(N for bwd and N for fwd) then calculating the actual gradient is trivial multiplication from equation 3.11. This improves the computation complexity in an order of magnitude($o(N^2)$ to $o(N)$) and the memory complexity by is still in the same order of magnitude($o(N)$ to $o(3N)$).

It's important to note that c is not the fidelity, but the overlap. We can get the fidelity

from c like so [8]

$$F = |c|^2$$

since c might be complex this derivative is a bit less trivial than it might look like. We can write $c(\vec{\epsilon})$ as $a(\vec{\epsilon}) + b(\vec{\epsilon})i$, where $a, b \in R$ and we get that

$$\frac{\partial F}{\partial \epsilon_k} = \frac{\partial |c|^2}{\partial \epsilon_k} = \frac{\partial ||a+bi|^2}{\partial \epsilon_k} = \frac{\partial (a^2+b^2)}{\partial \epsilon_k} = 2(a\frac{\partial a}{\partial \epsilon_k} + b\frac{\partial b}{\partial \epsilon_k})$$

We can notice that[9] $c(\frac{\partial c}{\partial \epsilon_k})^* = a\frac{\partial a}{\partial \epsilon_k} + b\frac{\partial b}{\partial \epsilon_k} + (ab - \frac{\partial a}{\partial \epsilon_k}\frac{\partial b}{\partial \epsilon_k})i$, more importantly we can see that the real part of that expression is exactly what we need, putting it all into one formula we get

$$\frac{\partial F}{\partial \epsilon_k} = 2 \cdot \mathrm{Re}\left\{ c(\frac{\partial c}{\partial \epsilon_k})^* \right\} \tag{3.12}$$

Now all you need is to plug 3.11 and 3.9 into 3.12 and you got your gradient :)

Let's do a little test now to see that everything is working well. The simplest pulse you can send is the pulse that takes the qubit from being in state $|0\rangle$ to state $|1\rangle$. We discussed this situation in appendix A so we know how the solution should look like. Running our GRAPE code with some random initial pulse we get

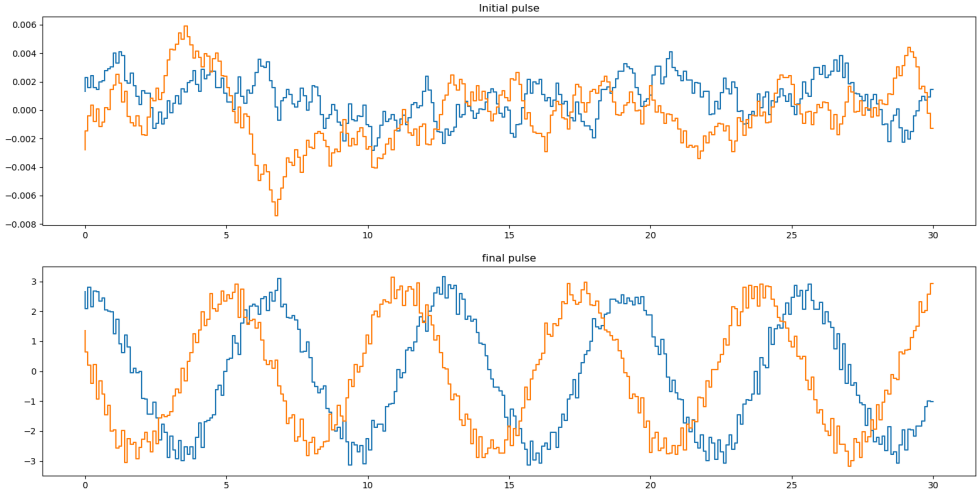


Figure 3.2: First example of GRAPE from state $|0\rangle$ to state $|1\rangle$

Amazing! From some random initial pulse we got sin and cos waves just as predicted in

[8] See initial definitions of c and the fidelity(3.9 and 3.1 respectively)

[9] $(\frac{\partial c}{\partial \epsilon_k})^*$ is the complex conjugate of the derivative of c

appendix A.

Before we get too excited there are a couple of things weird with this pulse. The first, more obvious problem, is that although the waves are sin and cos as expected, they're still pretty jagged-y, there is some randomness on top of the wave and it's not as smooth as we expected. This is since small random changes don't really change the final result don't really change the final result[10]. For reasons discussed in the next section we would like the pulse to be smooth.

Another problem that is not obvious right away we can see if we look at a graph of the level population over the duration of the pulse. We expect the graph to start at 1 and end at 0 for $|0\rangle$ and start at 0 and end at 1 for $|1\rangle$. Let's look at that graph for the initial random pulse and for the optimized pulse

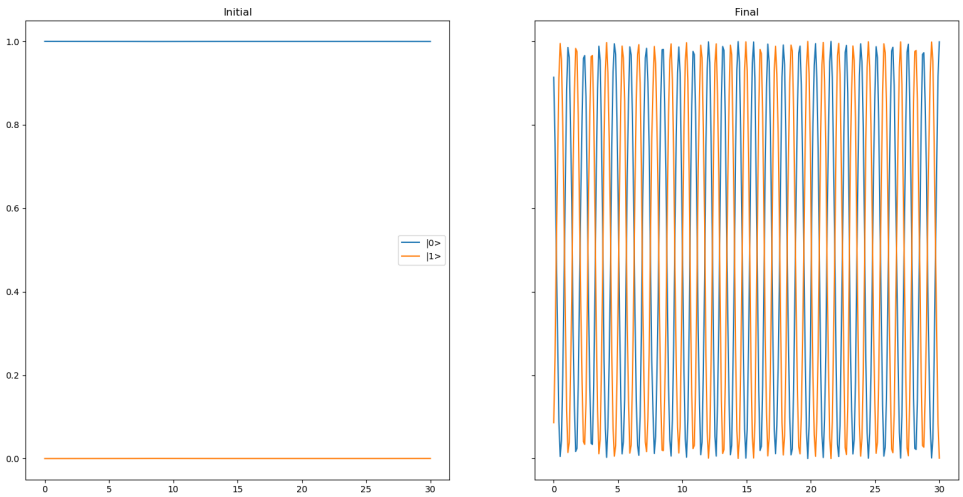


Figure 3.3: Population of qubit levels over pulse duration

As expected, the initial random pulse doesn't change the pulse almost at all. The optimized pulse on the other hand, WOW! What is this? It jumps between 0 and 1 too many times. Ideally, the population will change from 0 to 1(and vice-versa) smoothly only once.

This happens because the optimized pulse is wayyyy to strong, the amplitude of the pulse is huge! And well, as we defined the optimization, the algorithm doesn't care that the population does some weird stuff in the middle of the pulse as long as it ends at the

[10]The positive noise cancels out the negative noise

desired state.

To solve these problems(and more that we'll talk about later) we introduce *constraints* to the algorithm, as shown in great details in the next section.[11]

3.4 Constraints

Having our simulation doing whatever it wants is nice and all but still unfortunately, we're living in the real world, and in the real world we can't just make ultra-fast frequencies at close to infinite power, it's just impossible because of the limitation of our devices. We need to add constraints to the cost function so we won't get solutions that use too much power or that change too rapidly.

We define a set of constraints on the solution $g_i \geq 0$(ideally $g_i = 0$). We can associate a Lagrange multiplier λ_i to each constraint.
Now our goal is to maximize

$$F(\vec{\epsilon}) - \sum_i \lambda_i g_i(\vec{\epsilon})$$

Let's add a constraint to each of the most problematic physical limitation

3.4.1 Limiting the Pulse Amplitude

This is the most obvious physical limitation, we can't generate pulses with infinite energy, so we have to restrict it. There are two ways we can do so, the first is to create a hard cut-of amplitude, no matter what, the amplitude will never go above this amplitude, this will usually be the limit of our pulse generator. But normally we don't want our generator to work at it's absolute limit[12], so we can add also a soft amplitude maximum by "rewarding" the cost function to stay at a lower amplitude. Let's see how we would implement such a thing, starting with the hard cut-off.

[11] I'll give a quick note just to be honest, since this is such a simple case, GRAPE works pretty well even without any constraints. I've carefully crafted conditions so that the final pulse wouldn't be smooth and so the level population would go crazy. This is what you'll see normally in more complex examples but I didn't want to go with a complex example since it would just complicate things without giving any real benefit

[12] Not only that it might damage the device but also with stronger pulses the non-linear optics effects increase and then it's no fun :(

Instead of controlling and changing the amplitude($\vec{\epsilon}$) directly, we'll introduce a variable $\vec{x}$ and relate them as

$$\vec{\epsilon} = \epsilon_{max} \tanh \vec{x}$$

As you properly guessed, ϵ_{max} is the maximum amplitude of the pulse. Since the optimization algorithm can only change $\vec{x}$, the amplitude of the pulse will always be between $-\epsilon_{max}$ and ϵ_{max}. Unluckily for us, this changes the gradient of the cost function since we now want the derivative with respect to $\vec{x}$ instead of $\vec{\epsilon}$. We can relate the two

$$\frac{\partial F}{\partial \vec{x}} = \frac{\partial F}{\partial \vec{\epsilon}} \frac{\partial \vec{\epsilon}}{\partial \vec{x}} = \frac{\epsilon_{max}}{\cosh^2 \vec{x}} \frac{\partial F}{\partial \vec{\epsilon}}$$

We can use the derivative $\frac{\partial F}{\partial \vec{\epsilon}}$ we got from 3.12 and simply calculate $\frac{\epsilon_{max}}{\cosh^2 \vec{x}}$ and we again have the gradient.

For the soft limitation, there are two limitation we can make, linear and non-linear. The linear goal is for general preference of low amplitude pulses and the non-linear is if you have a specific $\epsilon_{max,soft}$ that you want to be well below of(You can use both or either one depends on your desired pulse properties). We'll start with the linear case since it's simpler.

For the linear amplitude penalty all we want is that *bigger amplitudes* $\Rightarrow$ *bigger cost function*, since our algorithm seeks to minimize the cost function, this will lead to the overall amplitude being smaller. The way we do so is simple, we can define a constraint $g_{amp,lin}$ that sums all the amplitudes of the steps of the pulse, so

$$g_{amp,lin} = \sum_k |\epsilon_k|^2$$

and we maximize the cost function as explained at the beginning of the section.

Still, since it is added to our cost function we need to find the gradient of the penalty as well. In the case it's rather simple since it's a basic parabola

$$\frac{\partial g_{amp,lin}}{\partial \epsilon_k} = 2\epsilon_k$$

and now we have all we need in order to add this penalty to our cost function. Let's move on to the non-liner penalty.

For the non-linear amplitude penalty we have a slightly different goal in mind

3.4.2 Limiting the Pulse Bandwidth and Slope

Again we encounter a physical limitation, we can't produce any pulse we want, there's a limitation of the maximum frequency our AWG(Arbitrary Waveform Generator) can create because the device can't change the voltage instantaneously. Again, like we had with the amplitude limit, there are 2 types of limits we can make, hard and soft. Let's start with the hard limit.

We have some frequency ω_{max} which is the maximum frequency that our AWG can generate. To make sure that our simulation doesn't produce such a pulse we can go from time space to the frequency space with a Fourier transform(Discrete Fourier Transform to be exact).

$$\vec{\epsilon} = (DFT)^{-1}\vec{x}$$

The numerical optimization algorithm controls $\vec{x}$ which is in the frequency space. Now if we want to limit the frequency we can simply set to 0 any frequency that is above our maximum frequency.[13]

$$x(\omega > \omega_{max}) = 0$$

The gradient of the new cost function is simply the Fourier transform of x

$$\frac{\partial \vec{x}}{\partial \epsilon_k} = (DFT)\frac{\partial \vec{\epsilon}}{\partial \epsilon_k}$$

And we know $\frac{\partial \vec{\epsilon}}{\partial \epsilon_k}$ from previous sections. It's important to note that the hard cut-off of the amplitude and the hard cut-off of the frequency do not work together since one is in the time space and one is in the frequency space. This is not much of a problem since we can compensate with the soft limits that do work well together(mainly since they require adding to the cost function instead of changing coordinate). In my simulations I use the amplitude hard limit instead of the frequency one since it only requires "squishing" the function instead of going from time space to frequency space.

[13]Might be more efficient to set these frequency to 0 and not let the optimization algorithm to try to change them(living the completely out of the pulse then putting them back in as 0 after the optimization is finished)

For the soft limits, we're limiting the slope(derivative) of the pulses and not the frequency directly. There are, again, two types of limits we can make, a linear limit and a non-linear limit. The linear limit simply incentives for a lower maximum slope overall, and the non-linear limit incentives a specific maximum soft limit on the slope, we can use the two together. Let's start with the simpler, linear limit.

The slope of a step function is simply $\epsilon_{k+1} - \epsilon_k$, we want to limit the size of the slope so we'll look at the expression $|\epsilon_{k+1} - \epsilon_k|^2$ instead. Summing all the slopes(to get an overall slope size of the entire pulse) we get the expression[14]

$$g_{slope,lin} = \sum_{k=0}^{N-1} |\epsilon_{k+1} - \epsilon_k|^2 \tag{3.13}$$

Remember that for each limit we associate a number $\lambda_{slope,lin}$ that is the "strength" of the limit, and that the cost function now becomes: $*old\ cost\ function* + \lambda_{slope,lin} g_{slope,lin}$, so we need to calculate the gradient of this limit and add it to the total gradient of cost function.

Unlike the amplitude, since the slope of the boundaries is not well defined we'll have the edges defined differently then the center of the pulse. The gradient of g in the center is a simple derivative, notice that each ϵ_k appears only twice in the sum

$$\frac{\partial g_{slope,lin}}{\partial \epsilon_k} = 4\epsilon_k - 2(\epsilon k + 1 + \epsilon k - 1)$$

It's nice to see that the expression looks like how'd you numerically estimate the second derivative, since the gradient of the slope(which is the derivative) is the second derivative, this is nice and reassuring :). Now we need to define the gradient at the edges, you can see that the derivative of ϵ_k depends on his +neighbors on both sides, since the first and last element of the pulse don't have 2 neighbors they are treated a little differently. each of the edges appears only once in the sum 3.13 unlike the others that appear twice, we

[14]Note that we have a problem at the edges since the slope of the end points is not well defined, we'll fix this problem later but for now we just ignore the last point $k = N$

can simply take the derivative of that one term and get

$$\frac{\partial g_{slope,lin}}{\partial \epsilon_0} = 2(\epsilon_1 - \epsilon_0)$$
$$\frac{\partial g_{slope,lin}}{\partial \epsilon_N} = 2(\epsilon_N - \epsilon_{N-1})$$

And now the slope soft linear limit is defined and so is it's gradient.

Now, before we continue to the non linear limit, we'll add another small constraint that will also solve the problem of the slope at the boundaries(you can think of it as a sort of boundary condition). It might seem weird at first, but we want to pulse to zero-out at the edges(the amplitude of the first and last steps of the pulse being equal to 0), this is since our AWG device can't immediately start a pulse with some amplitude, it can't get from 0 to that amplitude instantaneously(for the same reason we limit the slope in the first place). This could be achieved by simply setting the first and last steps of the pulse and their gradient to 0.

$$\epsilon_0 = \epsilon_N = \frac{\partial\ Cost}{\partial \epsilon_0} = \frac{\partial\ Cost}{\partial \epsilon_N} = 0$$

This solves the problem we were trying to solve we were having with the slope at the boundaries, since the gradient is 0 at the edges and does not depend on it's neighbors. We can move on to the non-linear limitation now.

As we mentioned earlier, the goal of the non-linear limitation is to get a specific wanted maximum slope instead of making the slope as small as possible, this is, like in the

3.4.3 Limiting Pulse Duration

As much as I wouldn't mind waiting a few nanoseconds longer for the qubit operation to end, the qubit itself isn't as patient as me. A state-of-the-art qubit would last, at most, one second(and that's a very conservative estimation), we simply don't have the time to wait for the operation to end if we want to run some complicated quantum circuit. This is why we want to add a constraint on the duration of the pulse, that way, if we give the pulse 5ns to finish, and a solution of 3ns exists, the optimization algorithm would(hopefully) find the 3ns solution and the rest of the time until the end of the duration will do nothing.

The constraint is fairly straight forward, add a penalty for any time a fidelity of 1 isn't achieved. Put into an equation we get

$$g_{duration} = \sum_{i=0}^{N-1}(1 - F_i)$$

Where F_i is the fidelity at time step i.

We can rather simply calculate the fidelity at any given time since the current way we calculate the fidelity, the state of the qubit at each time step is calculated(although not used, only calculated for the sole purpose of calculating the state at the next time step), we can simply modify the loop that calculates the final state into giving the fidelity at each time step and sum the results.

Luckily for us, the calculation of the gradient is also pretty simple, the gradient of the fidelity at each time step is calculated the same as the gradient of the fidelity we calculated in the beginning of the chapter[15]

$$\frac{\partial g_{duration}}{\partial \epsilon_k} = -\sum_{i=k}^{N-1} \frac{\partial F_i}{\partial \epsilon_k}$$

The calculation of the gradient

The pulse duration constraint works nicely to complete the other constraints. Without this constraint, the pulse will "try" to use all it's time to get the result we desire, and when running the algorithm without the constraint we can get problems if the duration we gave to the pulse is too long or too short. With this constraint on, we can simply give the algorithm a duration that we know for sure is more then the minimum required time and the algorithm will simply use the minimum time it need and no more. On the other hand, if we didn't have the amplitude(and bandwidth) constraints, the algorithm might find that it's best to just give a huge pulse for a tiny amount of time, but that's not physically possible as we discussed. This is why we can think of the constraints working together to "box in" the pulse into an ideal size.

[15]note that ϵ_k only appears in the expression for F_i, if $i > k$, so the sum starts at $i = k$

3.5 Implementing Qubit Operations with GRAPE

3.5.1 DRAG - Imperfect Qubits

When we did all of our calculation on the qubit we didn't include one detail, it's really hard to create a qubit, there's a reason why I don't have my quantum laptop yet after all :(. In the way our qubits are implemented, there are actually more then 2 levels. It's not a 2 level system but we treat it as one since the higher levels are off-resonance[16], but still there's a chance some of the higher levels will get excited by our pulses or some other physical phenomena and we want to account for it. The way we can do so is with the Derivative Removal via Adiabatic Gate(DRAG) algorithm. The idea being we make the qubit in the simulation to be more then a 2 level system, change the Hamiltonian a little bit(as you'll see later) so it accounts for the off-resonance higher levels and use the same grape algorithm to optimize for the entire system and not only the lower 2 levels.

Before we continue to implement DRAG, let's see if the 3rd level really is that of a problem, we'll run a simulation of GRAPE just as we did before but this time with 3 levels instead of 2, and the 3rd level should start and end at 0 population. We get after running GRAPE

[16] as explained in the beginning

Figure 3.4: Optimal pulses and level population with the DRAG method

Well yes, the forbidden layer did start and end at 0 population, but in the middle the qubit really became a 3 level system with the population of the forbidden level being almost $\frac{1}{2}$! This isn't one of the first two levels, we want to treat the system as if there are only two levels and the forbidden level should be negligible. More then that, the solution is noisy and not as smooth as we want, that is since it does the state transfer in a really unconventional manner, if we gave the algorithm a penalty on the forbidden level we would get a more conventional solution(sinusoidal).

We can't simply replace the qubit with a 3 level system and make the target of the third level always 0 and call it a day. We can't treat higher levels as another qubit level, they

are unwanted and we need to give them a penalty so the probability of being in a higher level would be always almost zero and change only a tiny bit. There are many ways we could implement such a penalty, the most obvious way is by simply making the probability to be in an higher level into a penalty, summing over all time we get(We'll call the third level of the qubit $|f\rangle$ to not be confused with the $|3\rangle$ Fock state(photon number state))[17]

$$g_{forbidden} = \sum_{i=0}^{N-1} \left| \left\langle f \middle| \psi_{fwd}^{(i)} \right\rangle \right|^2$$

We already have $\psi_{fwd}^{(i)}$ that we calculated earlier, so for so good.

Now moving to to complex part of DRAG, the gradient. Let's again define the overlap

$$c_f = \sum_{i=0}^{N-1} \left\langle f \middle| \psi_{fwd}^{(i)} \right\rangle$$

now to calculate the gradient we'll derive over ϵ_k

$$\frac{\partial c_f}{\partial \epsilon_k} = \frac{\partial}{\partial \epsilon_k} \sum_{i=0}^{N-1} \left\langle f \middle| \psi_{fwd}^{(i)} \right\rangle = \sum_{i=0}^{N-1} \frac{\partial}{\partial \epsilon_k} \left\langle f \middle| \psi_{fwd}^{(i)} \right\rangle$$

recall that $\psi_{fwd}^{(i)} = U_i \cdot U_{i-1} \cdot ... \cdot U_1 |\psi_{initial}\rangle$, U_k only appears for $i > k$, so we can start the sum from $i = k$. We'll also expand $\psi_{fwd}^{(i)}$ into what it is and get

$$\frac{\partial c_f}{\partial \epsilon_k} = \sum_{i=k}^{N-1} \frac{\partial}{\partial \epsilon_k} \langle f| U_i \cdot ... \cdot U_0 |\psi_{initial}\rangle$$

the only element that's dependent on ϵ_k is U_k, so we can rearrange the equation as

$$\begin{aligned} \frac{\partial c_f}{\partial \epsilon_k} &= \sum_{i=k}^{N-1} \langle f| U_i \cdot ... \cdot \frac{\partial U_k}{\partial \epsilon_k} \cdot ... \cdot U_0 |\psi_{initial}\rangle \\ &= \sum_{i=k}^{N-1} \langle f| U_i \cdot ... \cdot i \cdot \delta t \frac{\partial H_k}{\partial \epsilon_k} U_k \cdot ... \cdot U_0 |\psi_{initial}\rangle \\ &= i \cdot \delta t \sum_{i=k}^{N-1} \langle f| U_i \cdot ... \cdot U_{k+1} \cdot \frac{\partial H_k}{\partial \epsilon_k} \left| \psi_{fwd}^{(k)} \right\rangle \end{aligned}$$

[17] If we wanted to accounted for higher levels we can sum over the sum for each level

Now just to keep everything simple and maintainable, we'll define

$$\left\langle \phi_{bwd}^{(i,k)} \right| = \langle f | \, U_i \cdot ... \cdot U_{k+1}$$

The equation for the overlap now becomes

$$\boxed{\frac{\partial c_f}{\partial \epsilon_k} = i \cdot \delta t \sum_{i=k}^{N-1} \left\langle \phi_{bwd}^{(i,k)} \right| \frac{\partial H_k}{\partial \epsilon_k} \left| \psi_{fwd}^{(k)} \right\rangle}$$

Now we got all we need to calculate the penalty of the occupying the higher level and it's gradient. This isn't a perfect solution though, for N time steps we need to do $o(N^2)$ calculations to get $\left\langle \phi_{bwd}^{(i,k)} \right|$, this slows down the calculation considerably[18] and there is a lot of overhead in the way we calculated $\left\langle \phi_{bwd}^{(i,k)} \right|$. We can use a smarter way to calculate it.

Consider a function, very similar to $\langle \psi_{bwd}|$ we had earlier(in fact, it's the same function minus multiplying by the target state on the left)

$$\psi_{bwd}^{(k)} = U_N U_{N-1} ... U_{k+2} U_{k+1}$$

taking the inverse of the resulting matrix we get

$$(\psi_{bwd}^{(i)})^{-1} = U_{i+1}^{-1} U_{i+2}^{-1} ... U_{N-1}^{-1} U_N^{-1}$$

by multiplying the two matrices we get(defining their product as ϕ_{bwd})

$$\phi_{bwd}^{(i,k)} = (\psi_{bwd}^{(i)})^{-1} (\psi_{bwd}^{(k)}) = (U_{i+1}^{-1} \cdot ... \cdot U_N^{-1})(U_N \cdot ... \cdot U_{k+1}) = U_i \cdot ... \cdot U_{k+1}$$

This is exactly what we wanted! from this we'll define

$$\left\langle \phi_{bwd}^{(i,k)} \right| = \langle f | \, \phi_{bwd}^{(i,k)}$$

remember that ψ_{bwd} was already calculated from the gradient calculation, taking the inverse of ψ_{bwd} isn't affected by how many time steps there are, also the multiplications between ψ_{bwd}, $(\psi_{bwd})^{-1}$ and $\langle f|$ isn't dependent on the amount of time steps, so the entire

[18] it makes to calculation run around 100 times slower, pretty bad considering it's just a penalty

calculation is $o(1)$ complexity. We went from $o(N^2)$ to $o(1)$ with this simple trick!

Let's run now the algorithm and get some results

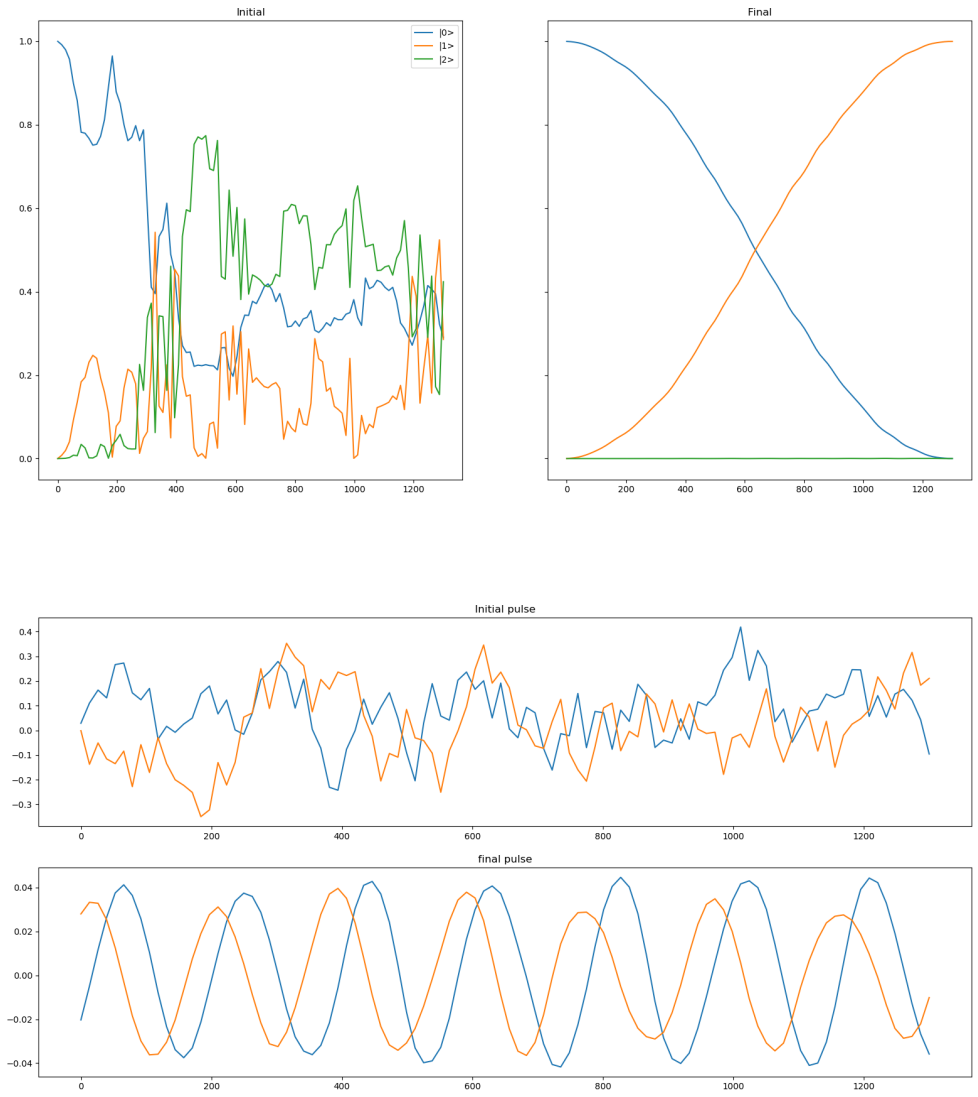


Figure 3.5: Optimal pulses and level population with the DRAG method

Nice! state $|0\rangle$ goes directly to 1, $|1\rangle$ goes directly to 0 and the forbidden level is barley changed throughout the pulse(the fidelity gotten from this pulse is around 99.9%, so pretty good).

I think that we talked enough about the qubit for now, let's move the the other half of the system, the cavity.

3.6 Implementing Cavity Operations

3.6.1 Limiting the photon number

"Hilbert space is a big place."

- Carlton Caves

Here's the thing about the cavity levels, there are infinite amount of them. This might be a problem since our computers can't really deal with infinite amount that well. We can make an assumption that the cavity only has N levels but it is still possible that something happens in the higher levels that may affect the physical result that we didn't include in the simulation. We want to limit that and make sure that everything interesting is contained in the N levels that we have.

This is quiet similar to what we did in the previous section, we want to put a penalty on the higher levels, still, there are two main differences. The first, is that there are much more then one or two extra levels, and as we've seen, the method we used in the previous uses very heavy computation and we can't do it for so many levels since we want our computer to not explode. The other difference is that care less about if some higher level is occupied for a part of the pulse, in the cavity there are higher levels and they're all likely to be but the reason we're limiting the cavity levels is for computing reasons not physical ones. Unlike the cavity, we really want the qubit to have only two levels and the only reason we simulate the higher ones is because the physical world is not as fun as the mathematical one and we can't simply ignore the higher levels. So it makes sense that we'll use a different penalty to limit the photon number. Let's take a look on how we'll do so.

The idea is this, we'll define n_{ph} as the highest level we want the cavity to have, now let's calculate what will happen if the cavity will have another level, for a total of $n_{ph} + 1$. Ideally, nothing will change, the new level should start at 0 probability and end at 0 probability with no change in between, if there is a change, will add a penalty to the pulse, so in the next iteration there will be less of a change. We can do so for and level higher then 1, so instead of using only $n_{ph} + 1$ we'll some over $n_{ph} + k$ for reasonable amount of k's.

We'll define $F_{n_{ph}+k}$ as the fidelity if there were $n_{ph}+k$ levels. Putting the idea into a formula we get that the new cost function is given by

$$Cost = \sum_{k=0}^{N} F_{n_{ph}+k}(\vec{\epsilon}) - \sum_{i} \lambda_i g_i(\vec{\epsilon})$$

We can double enforce the penalty if we add a constraing making sure there is no change in the fidelity for different levels

$$g_{ph} = \sum_{k_1 \neq k_2} (F_{n_{ph}+k_1} - F_{n_{ph}+k_2})^2$$

The gradient of which is simply the gradient of which is simply

$$\frac{\partial g_{ph}}{\partial \epsilon_k} = 2 \sum_{k_1 \neq k_2} [(F_{n_{ph}+k_1} - F_{n_{ph}+k_2})(\frac{\partial F_{n_{ph}+k_1}}{\partial \epsilon_k} - \frac{\partial F_{n_{ph}+k_2}}{\partial \epsilon_k})]$$

and everything in this expression was previously calculated(they're simply the derivatives of the fidelity and the fidelity itself).

It's important to note that while it might be tempting to leave n_{ph} at a small value so there will be less to calculate(the size of the matrices grows with n_{ph}^2), there is good reason to use a high values of n_{ph}. Bigger n_{ph} oscillate at higher frequency(since the cavity is simply a harmonic oscillator), so it's possible to use shorter pulses, and since keeping qubit alive is really a major problem, keeping the pulses short is important to be able to accomplish the most with the time we have with the qubit before it dies.

3.7 Finding a Good Initial Guess

Although the GRAPE algorithm is the one responsible to find to optimal control pulse, we still need to give it some initial guess and the algorithm does the rest. You might think that this isn't much of a problem since theoretically any initial guess should arrive at a desired result. The problem is that many times the algorithm gets stuck and can't find a result. This could be caused by a number of reasons, the main two are when the constraints are too strong and when the initial guess is not good enough.

When the constraints are two strong, the algorithm might prefer optimizing them instead

of the fidelity and what we get is a pulse that achieves horrible fidelity but within the constraints. This is really not good and the solution is simply weakening the constraints(choosing a smaller λ for the constraint).

The other, harder to solve problem is when the initial guess of the pulse is problematic. For example, if you choose the initial guess the be the most obvious initial guess you can make, constant 0, the algorithm will properly stop after one iteration changing nothing, this is because the gradient of a constant 0 pulse is actually zero, so the optimization algorithm think it's in an optimized minimum when in fact, it properly couldn't be more wrong.

The other simple initial guess you might think of using is a random pulse, after all we don't know what is the desired pulse so picking a random pulse is as good as any other, plus it has the added benefit that even if one random pulse doesn't get you to a solution, the next time you run it it might get to one. The problem with a random pulse is that it is the definition of a non-smooth function, and since the initial guess is not smooth, the algorithm might find it difficult to get a smooth solution.

There are two approaches we can take to get a good initial pulse, the first approach assumes nothing about the system, this is good since it is really general and can be used in any case, but might be not ideal in some cases. The second approach is when we know roughly how the solution should look like, we can use some pulse that is similar to what we expect and GRAPE will get the actual pulse from that.

In the first aproach, we want GRAPE to do most of the work, but not get stuck by some weird problem of the initial pulse, we want a guess that is close to 0, pretty random, but not so much that it would be hard to smooth. We can get such a pulse by doing a *convolution* between a random pulse and a Gaussian window

Unlike the first approach, the second approach could look really different for different examples but I'll give some general guidelines you can use the get a good initial pulse. Let's say, for example, you have a 3-level system where the third level isn't wanted(such as in the DRAG example). If you give an initial guess like the one in the first approach, the third level will still be excited by that pulse, and it might be hard for the algorithm to fix this. What you might do in this scenario, is to start with an initial guess that you know

excites only the first and second levels of the system but not the third. This is easy since you know the energy difference between the levels, and from that you know the frequency that excites each level. What you can do is some random pulse that has frequency around the first-second levels frequency difference. So if you look in the frequency space, what you see is some Gaussian distribution around the first levels frequency with some random noise on it.

3.8 From States to Gates

Until now we've discussed how to find pulses that take our system from one state to another. Which raises an important question, ***why****?*. Sure, creating Fock states in cavity is nice and all, but I want to run quantum algorithm on my quantum computer not make pretty pictures.

Here's the thing, turns out, you can change the algorithm a little bit and get a GRAPE algorithm that gives you back the optimal pulses that realizes a desired *quantum gate*, instead of just taking you from one state to another. Let's take a look on how this is done.

4 Controlling a Superconducting Quantum Computer

4.1 Overview

Before jumping into the specifics of how'd we control the quantum computer, let's start with a general overview and show how everything is connected and the purpose of it all.

We'll start with a diagram that shows how the system is connected, from the pulse generator to the cavity.

4.2 Generating the Pulses

4.2.1 The AWG

It shouldn't be too much of a surprise if I'll tell you that I consider the AWG(Arbitrary Waveform Generator) the "heart♡" of the system, we've just spent an entire chapter on finding the pulses we want to send, and the instrument that creates and sends the pulse, is the AWG. Unfortunately, using the AWG isn't as straight-forward as you might hope and there are some problem's we'll need to address if we want our system to work.

Consider for a moment what we want to do with our microwave generator to control a qubit, we need to send a high frequency signal and change it by a really small amount compared to its frequency(sending a GHz signal and changing it by MHz, 3 orders of magnitude difference). We can't just take a wave generator that generates a 10GHz signal and tell it to change from 10GHz to 10.0001GHz because it won't be precise enough, and even if the frequency generator is good enough so that we can make really accurate changes to the frequency, we still want to have the wave interact with the quantum system and analyze the result so we also need a super-frequency-analyzer to understand what the hell is going on in the quantum system, not only that, we also want the frequency analyzer and generator to work together(matching the same input with the corresponding

output of the system). Now that we understand what are some of the challenges working with such high frequencies, what can we do about it?

Well the solution is simple, we can take a high frequency wave(10GHz for example) and a lower frequency wave(1MHz for example) and mix them together *somehow* to get a 10GHz + 1MHz = 10.0001GHz signal, and in the output of the quantum system we can simply un-mix the result to get back the 10GHz signal and some other signal(in the MHz) that can tell us something about the system.

That sounds all nice and good but how do we actually mix/un-mix 2 signals in that way and how can we control that proccess? That's where the IO-Mixer comes in, we'll look into them in just a moment, first we need to understand a regular mixer work.

4.2.2 The Mixer

The idea is simple, the mixer has 2 inputs and one output, when you enter 2 waves as an input, you get their product as the output(inputting for example $\cos(t)$ and $\cos(2t)$ will result in $\cos(t)cos(2t)$ at the output). We draw a mixer in a diagram like so,

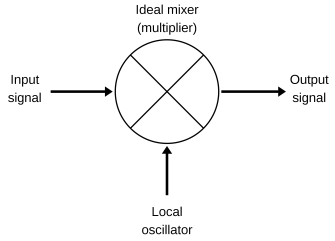


Figure 4.1: Ideal mixer in a diagram

We can change what are the outputs and what are the inputs to get different ways for the mixer to work[19]

4.2.3 The IQ-Mixer

We've seen what's a *regular*(and *ideal*) mixer is, but how can we use it for the desired effect? remember, we want to input a high frequency and a lower frequency and we want

[19]TODO: Explain this

the output to be a wave with a frequency that is the sum of the 2 frequencies. To do so, we can consider the following diagram

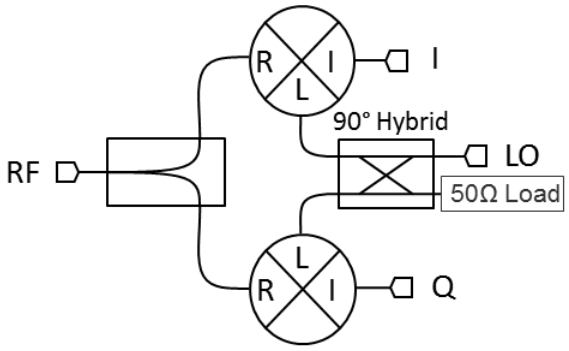


Figure 4.2: The IQ mixer

Where the 90° hybrid in the diagram is a *90° hybrid coupler*, it splits the signal into 2 signals at a 90°phase difference, hence the name. The square near the RF sign simply adds the 2 waves, we also look into it in the appendix

As we can see, the IQ mixer has 3 inputs, I, Q(hence the name) and LO. We can also see that there's only one output, RF(although you can play with what are the inputs and what are the outputs).

How can we use this IQ mixer to add frequencies? Let's consider the following inputs[20]

$$I - -- > \cos(\omega_{IQ}t)$$
$$Q - -- > \sin(\omega_{IQ}t)$$
$$LO - -- > \sin(\omega_{LO}t)$$

In this case, the input into the top mixer will be I and a LO, which is $\cos(\omega_{IQ}t)\sin(\omega_{LO}t)$. Similarly, the input into the bottom mixer will be Q and 90°phase of LO, which is $\sin(\omega_{IQ}t)\cos(\omega_{LO}t)$.

The total output(in RF) will be the sum of the two waves

$$RF = \cos(\omega_{IQ}t)\sin(\omega_{LO}t) + \sin(\omega_{IQ}t)\cos(\omega_{LO}t)$$

and we know from simple trigonometry that $sin(a+b) = \cos(a)\sin(b) + \sin(a)\cos(b)$, so

[20]You can flip I and Q and get subtraction instead of addition

we get

$$\boxed{RF = \sin((\omega_{LO} + \omega_{IQ})t)} \tag{4.1}$$

Perfect! this is exactly what we wanted, the output is a wave with frequency that is the some of the input frequencies. Only one problem, it doesn't work :(.

4.2.4 Theory VS Reality :(

If we use a spectrum analyzer and view what frequencies are in the final wave we get the following picture

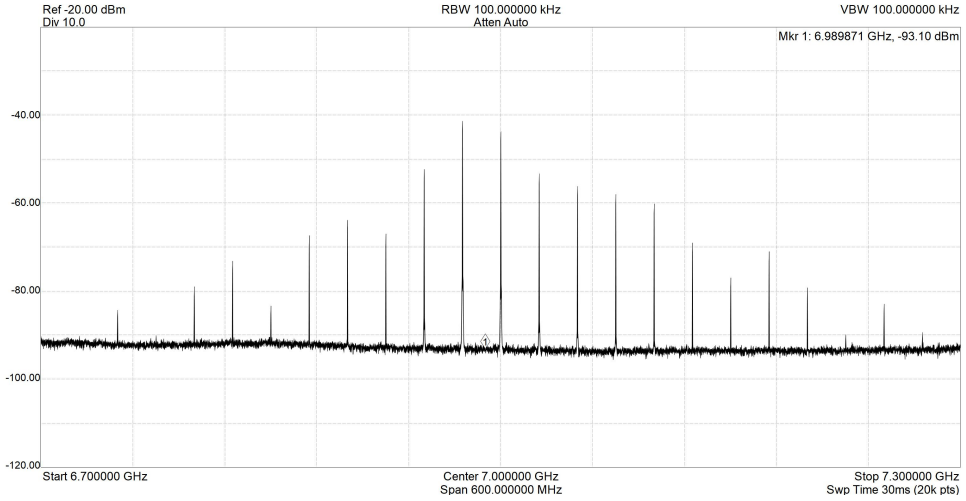


Figure 4.3: Full Spectrum Without Any Corrections

Zooming in around the LO frequency we see

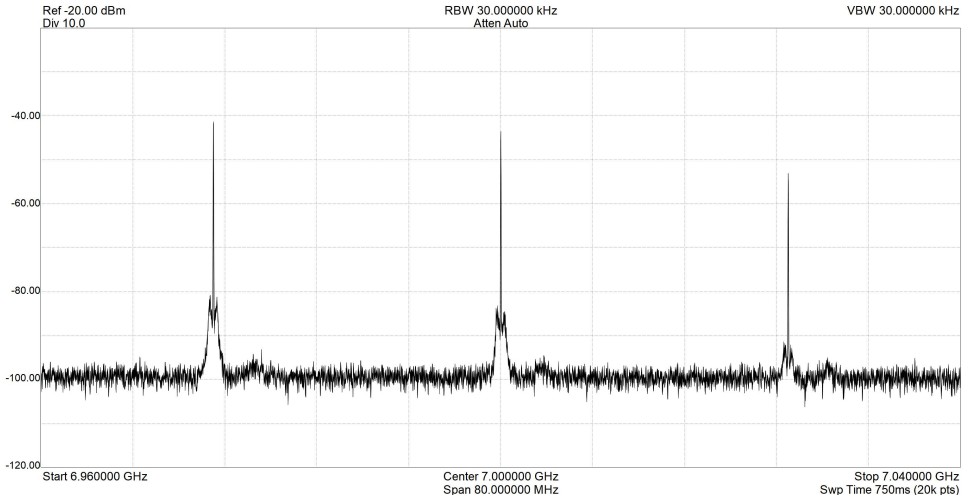


Figure 4.4: Spectrum Around the LO Frequency

What is the problem? We've proved mathematically that it should work, so why doesn't it? The problem is that we can't just assume the waves to come and go with the same phase, the waves travel through the wire at some speed so if we input into two different wires, two waves that are at the same phase, at the other side of the wire they might come at different phases because of differences in wire length, resistance, etc... So what can we do about it? You could try to make identical parts and make everything just perfect but even slight deviation will cause the system no to work properly, a better solution is to input more complex waves and have some parameters to play with so we can simply find the right parameters for the system and then it will work.

We can analyze the frequency space of the output of out IQ mixer and we can see 2 types of it not working correctly

- Leakage at the LO frequency
- Leakage at $\omega_L O - \omega_I Q$

We can solve the first type of Leakage, at the LO frequency, by adding DC offsets to the input frequencies(We'll prove this mathematically later), and we can solve the second type of leakage by adding phase offsets to the waves(We'll prove this mathematically later).

4.2.5 Solution for the real world

Before we can solve the problem, we need to understand what's causing it. As we explained earlier, a phase is created in the wires that connect everything.

Leakage at $\omega_{LO} - \omega_{IQ}$

Let's consider now inputting into the IQ mixer the same waves but with the phases that were created in the transmission wires instead of what we had earlier

$$I - - - > \cos(\omega_{IQ} t + \phi_I)$$

$$Q - - - > \sin(\omega_{IQ} t + \phi_Q)$$

$$LO - - - > \sin(\omega_{LO} t + \phi_{LO})$$

Using the same calculation we did before, we get that

$$RF = I \cdot LO + Q \cdot LO(90°)$$

$$RF = \cos(\omega_{IQ}t + \phi_I) \cdot \sin(\omega_{LO}t + \phi_{LO}) + \sin(\omega_{IQ}t + \phi_Q) \cdot \cos(\omega_{LO}t + \phi_{LO})$$

If you can trust me on the algebra(or rather, trust wolfram alpha on the algebra...) we get the expression

$$\begin{aligned} RF = &\cos\left(\frac{\phi_Q - \phi_I}{2}\right) \sin\left((\omega_{IQ} + \omega_{LO})t + \frac{\phi_Q + \phi_I}{2} + \phi_{LO}\right) \\ &+ \sin\left(\frac{\phi_Q - \phi_I}{2}\right) \cos\left((\omega_{IQ} - \omega_{LO})t + \frac{\phi_Q + \phi_I}{2} - \phi_{LO}\right) \end{aligned}$$

This expression is quiet scarier than the one we got earlier... More than that, we get two frequencies instead of one, we've got an unwanted frequency at $\omega_{LO} - \omega_{IQ}$ and the only way to make it disappear is if the phases are equal, $\phi_I = \phi_Q$. Also the final wave as a phase of ϕ_{LO}, this isn't really a problem and if we define our starting point differently we can set ϕ_{LO} to 0.

LO Frequency Leakage

Another type of leakage we've observed is leakage at the LO frequency, it makes sense that some of the original wave will go through the mixer untouched. To fix that leakage, we'll need to somehow change the I and Q waves to cancel it out. The simplest way to do so is to add DC offsets to the inputs [21]

$$\begin{aligned} I &--> \cos(\omega_{IQ}t + \phi_I) + \epsilon_I \\ Q &--> \sin(\omega_{IQ}t + \phi_Q) + \epsilon_Q \\ LO &--> \sin(\omega_{LO}t) \end{aligned}$$

[21] I've removed the phase on the LO wave since we've seen it doesn't really change anything

We've seen this story before... the calculation of the RF wave is the same so we'll skip the calculation. The end result is

$$\begin{aligned} RF =& \cos\left(\frac{\phi_Q - \phi_I}{2}\right)\sin\left((\omega_{IQ} + \omega_{LO})t + \frac{\phi_Q + \phi_I}{2}\right) \\ &+ \sin\left(\frac{\phi_Q - \phi_I}{2}\right)\cos\left((\omega_{IQ} - \omega_{LO})t + \frac{\phi_Q + \phi_I}{2}\right) \\ &+ \epsilon_I \sin(\omega_{LO}t) + \epsilon_Q \cos(\omega_{LO}t) \\ =& RF_{old} + \epsilon_I \sin(\omega_{LO}t) + \epsilon_Q \cos(\omega_{LO}t) \end{aligned}$$

where RF_{old} is the RF wave before adding the DC offsets.

What we get is the same wave, but now we can play with the LO frequency at the output. Later we'll change the DC offsets so that they will cancel to LO frequency leakage to minimize it.

Now that we have all of our "knobs" we can change and play with, we can start using them to minimize the leakages.

4.2.6 Finding Optimal Constants

As we've seen in the previous section, there are 4 variables we can "play" with to get the best variables for our system, as long as we don't change the system, these variables stay the same. What we want to do now is to actually find them. Our system is connected like so

We have the Quantum Machine[22] that generates the I and Q inputs that go into the mixer(and also to an oscilloscope for debugging). There's the frequency generator[23] that is connected to the mixer and generates 7GHz wave, and there's the frequency spectrum analyzer[24] that is connected to the computer.

Let's first attack the leakage at the LO frequency. For now we'll have an LO frequency of 7GHz that we want to change by 25MHz(The same variables work for all frequencies this is just as an example)

[22]this is the heart of the system, for now we'll use it to make the MHz waves with different phases, DC offsets and frequencies

[23]KeySight N5173B

[24]SignalHound USB SA-124B

4.2.6.1 LO frequency Leakage

As proven in section 4.2.5, to minimize this kind of leakage all we need is to play with the DC offsets of the IQ inputs. To do so, we first need to define what we want to minimize, which in this case is simply the power of the frequency at 7GHz, we can measure that power with our spectrum analyzer, we'll call that our *cost function*. We have a 2-dimensional variable space, we need to find where in this space the cost function is at a minimum. To do so, we'll start by using a brute force method to find the general location of the minimum in the variable space, since brute force is very inefficient we can't really use it to find the exact location of the minimum so we start by only doing a low precision brute force and then use a different optimization algorithm to find the exact location of the minimum. We'll use the `scipy.optimize.fmin` as the algorithm for precise minimum location.

4.2.6.2 Harmonies Leakage

Now that we've minimized the LO frequency leakage, we want to minimize the harmony leakage, we do that by changing to phases of the IQ waves from the quantum machine, it's important that changing the phases doesn't change the LO leakage and luckily for us, as proven in section 4.2.5 that's whats happening. to change the phases we don't simply specify the phases, we use the correction matrix of the Quantum Machine. This time our cost function is the frequency at $\omega_{LO} - \omega_{IQ}$, we can do the same as we did in the LO leakage and use first a brute force optimization to find the general location of the minimum and the use the fmin algorithm to find the exact location of the minimum of the cost function(this time in the scale-angle variable space). You can see the result here

4.3 Some other problems maybe?

...

5 Future Work and Conclusions

...

A Analytical Calculations of Optimal Pulses

Throughout chapter 3 we embarked on journey finding optimal pulses with numerical methods, but it's important to note that in some specific cases we can calculate the solution analytically. This has more uses than for mathematical beauty, we can use these cases as test cases to see that the GRAPE algorithm actually works and gives the same answer as we got in the analytical calculation.

B Another Method: Computational Graphs

In the optimal control section, in essence, we try to minimize the cost of many many variables, this is a similar problem that to teaching neural networks and machine learning, we might be able to borrow some tricks and method they use to solve our problem.

Instead of thinking of a cost function as just a function, we can treat it as a *computational graph*. We'll discuss briefly about what's a computational graph and how can we implement GRAPE using one. We won't go in depth as we did in the chapter on optimal control, it's only a general overview meant to show an alternative method. Let's start by defining a computational graph[25].

B.1 What are Computational Graphs

A computational graph consists of nodes and connection between them, a node can be one of one of three things

- **Operations**, the operation takes a list of numbers from other nodes and outputs another list of numbers(doesn't need to be of same size)
- **Parameters**, these are, as the name suggests, the parameters of the graph and can be used by the operations.
- **Variables**, these are the variables that of the graph, and they too can be used by the operators of the graph

The graph starts as the variables, goes through some operations that use parameters and gives out some resualt. Let's take a look at a simple example

[25]Of course, graph theory is an entire field of mathematics by itself and we can't really give a rigorous definition, but more of an intuitive explanation

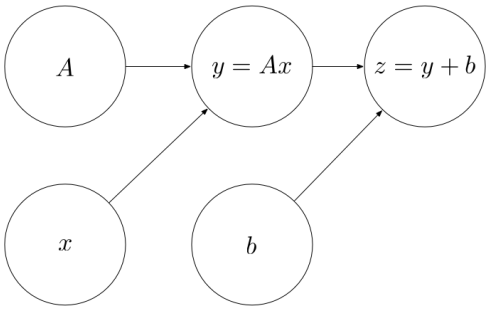


Figure B.1: Example of a basic computational graph

Here, x is the variable, A and b are the parameters, and y and z are the operators. The function that this graph represents is $y = a \cdot e^x$. For now it properly seems overkill to use the graph to represent that simple of a function but in the next two sections you'll see the magic of the computational graph, in terms of thinking about the gradient.

B.2 Back-Propagation

This is the magic of the computational graph, calculating the gradient of the cost function by back-propagating the derivatives to the initial variables. The idea is simple, instead of trying to directly calculating the gradient, we calculate the derivative of node by the previous node, and relate the cost function to the variables by the chain rule. What does that mean, let's show an example.

Let's say we have a simple comutational graph that looks like

$$x \to L_1 \to L_2 \to L_3 \to y$$

and we want to calculate the derivative, $\frac{\partial y}{\partial x}$. Instead of calculating the derivative directly, we can back-propagagte the derivatie as

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial L_3} \cdot \frac{\partial L_3}{\partial L_2} \cdot \frac{\partial L_2}{\partial L_1} \cdot \frac{\partial L_1}{\partial x} \cdot \frac{\partial y}{\partial x}$$

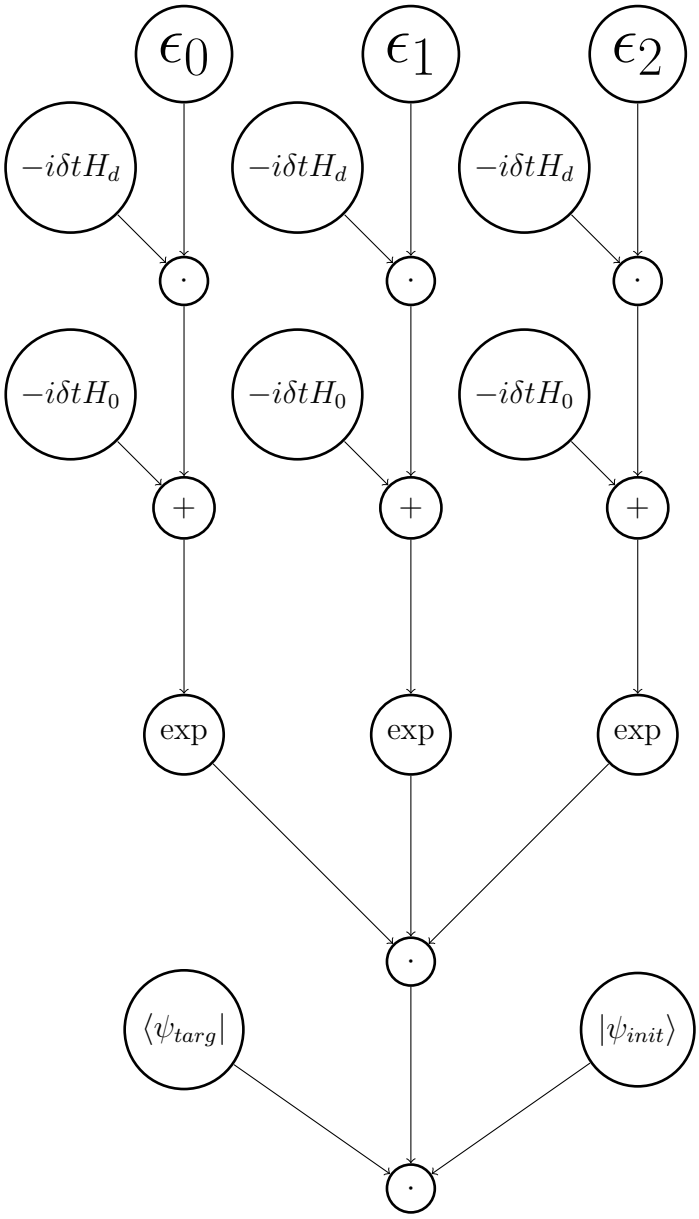
luckily for us each operation is very simple so each derivative is simple as well.

B.3 GRAPE in a Computational Graph

Now we can finally implement grape as a computational graph, we'll consider the simplest case of GRAPE, no constraints, only the original definition(see equations 3.8, 3.6 and 3.3)[26]

$$F(\vec{\epsilon}) = \left| \langle \psi_{target} | \prod_{i=0}^{N} e^{-i \cdot \delta t (H_0 + \epsilon_k H_d)} | \psi_{initial} \rangle \right|^2$$

In this case, $\vec{\epsilon}$ is the variable, H_0, $-i \cdot \delta t$, $\langle \psi_{target}|$ and $|\psi_{initial}\rangle$ are the parameters. They relate through the operators and everything can be displayed as the following graph[27]



[26] This is assuming only one drive hamiltonian, for more drives the only difference is adding $\epsilon'_k H'_d$ and so on for each drive

[27] Simplifying for 3 time steps, to add more time steps is pretty straight forward

The layers are as follows

$$
\begin{aligned}
L_1^k &= -i\delta t H_d \epsilon_k \\
L_2^k &= -i\delta t H_0 + L_1^k \\
L_3^k &= e^{L_2^k} \\
L_4 &= \prod_{k=0}^{N} L_3^k \\
C &= \langle \psi_{target} | \, L_4 \, | \psi_{initial} \rangle
\end{aligned}
$$

And the derivatives are calculated rather easily as

$$
\begin{aligned}
\frac{\partial L_1^k}{\partial \epsilon_k} &= -i\delta t H_d \\
\frac{\partial L_2^k}{\partial L_1^k} &= 1 \\
\frac{\partial L_3^k}{\partial L_2^k} &= e^{L_2^k} \\
\frac{\partial L_4}{\partial L_3^k} &= \prod_{i \neq k} L_3^i \\
\frac{\partial C}{\partial L_4} &= \overline{|\psi_{target}\rangle} \otimes |\psi_{initial}\rangle \;\; (Need\ correcting)
\end{aligned}
$$

It's important to note that $\frac{\partial L_1^i}{\partial \epsilon_j} = 0$ for $i \neq j$ so we didn't refer to those derivative(This is true for the derivative between any two layers).

We now everything we need to implement GRAPE as a computational graph, and we can use a library, such as google's tensorflow, to find the optimal pulse.