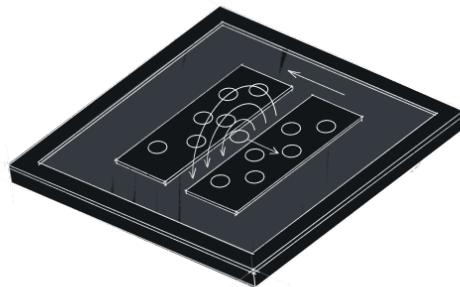


Quantum Processor Theory And Design

“There were several possible solutions of the difficulty of classical electrodynamics, any one of which might serve as a good starting point to the solution of the difficulties of quantum electrodynamics.” - Richard Feynman



Bartu Yaman

Contents

1	Introduction	8
1.1	Introduction To Electronics	9
1.1.1	Capacitors	9
1.1.2	Inductors	9
1.1.3	LC Circuits	10
1.2	Introduction To Quantum Mechanics	13
1.2.1	Quantum Harmonic Oscillator	13
1.2.2	Lamb Shift	14
1.2.3	Kerr Effect	14
1.2.4	Tunneling	14
1.2.5	Cooper Pairs (Bardeen- Cooper- Schrieffer pair)	15
2	Circuit Quantum Electrodynamics	16
2.1	Superconducting Quantum Circuits	17
2.1.1	The Quantum LC Resonator	17
2.1.2	2D Resonators	19
2.1.2.1	Quantized Modes Of Transmission Line Resonator	22
2.1.3	The Transmon Artificial Atom	22
2.1.4	Flux-Tunable Transmons	27
2.2	Light-Matter Interaction In Circuit QED	29
2.2.1	Exchange Interaction Between A Transmon And An Oscillator	29
2.2.2	The Jaynes-Cummings Spectrum	31
2.2.3	Dispersive Regime	32
2.2.3.1	Schrieffer-Wolff Approach and Bogoliubov Approach	32
2.2.4	Josephson Junctions Embedded In Multimode Electromagnetic Environments	35
2.2.4.1	Beyond the transmon: multilevel artificial atom	37
2.2.4.2	Alternative coupling schemes	38
2.2.5	Coupling To The Outside World: The Role Of The Environment	38
2.2.5.1	Wiring Up Quantum Systems With Transmission Lines	39
2.2.5.2	Input-Output Theory In Electrical Networks	41
2.2.5.3	Qubit Relaxation And Dephasing	43
2.2.5.4	Dissipation in the dispersive regime	45
2.2.5.5	Multi-mode Purcell effect and Purcell filters	49

2.2.5.6	Controlling quantum systems with microwave drives	49
2.2.6	Measurements in Circuit QED	51
2.2.6.1	Microwave field detection	51
2.2.6.2	Phase-space representations and their relation to field detection	56
2.2.6.3	Dispersive qubit readout	58
2.2.6.4	Signal-to-noise ratio and measurement fidelity	61
2.2.7	Qubit-Resonator Coupling Regimes	66
2.2.7.1	Resonant regime	67
2.2.7.2	Dispersive regime	72
2.2.7.3	Beyond strong coupling: ultrastrong coupling regime	77
2.2.8	Quantum Computing with Circuit QED	78
2.2.8.1	Single-qubit control	78
2.2.8.2	Two-qubit gates	81
2.2.8.3	C. Encoding a qubit in an oscillator	92
2.2.9	QUANTUM OPTICS ON A CHIP	96
2.2.9.1	Intra-cavity fields	97
2.2.9.2	Quantum-limited amplification	98
2.2.9.3	Propagating fields and their characterization	100
2.2.9.4	Remote Entanglement Generation	103
2.2.9.5	Waveguide QED	105
2.2.9.6	Single microwave photon detection	107
2.2.10	OUTLOOK	107
3	Quantum Processor Design	110
3.1	Overview	111
3.1.0.0.1	You'll use Qiskit Metal in 4 stages	111
3.1.0.1	This tutorial is for steps 1 and 2	111
3.1.0.2	Using this Tutorial	111
3.1.0.3	QDesign (need-to-know)	111
3.1.0.4	QDesign (in-depth)	112
3.2	Coding Time!	112
3.2.1	Import Qiskit Metal	112
3.2.2	My First Quantum Design (QDesign)	113
3.2.3	My First Quantum Component (QComponent)	114
3.2.3.1	A transmon qubit	114
3.2.3.1.1	What are the default options?	115
3.2.4	Closing the Qiskit Metal GUI	116
3.2.5	My first Quantum Design (QDesign)	116
3.2.6	My First Quantum Component (QComponent)	116
3.2.6.1	A transmon qubit	116
3.2.6.1.1	What are the default options?	118
3.2.6.2	Where are the QComponents stored?	118
3.2.6.2.1	Parsing strings into floats	120
3.2.6.2.2	Some basic arithmetic and parsing	120
3.2.6.3	Advanced: parse into arrays, list, etc.	121

3.2.7	How do I overwrite QComponents?	122
3.2.7.1	QPins: The dynamic way to connect qcomponents	122
3.2.7.2	How do I edit the component source code and see changes immidiately?	123
3.2.8	Creating a whole chip of qubit with connectors	123
3.2.9	Variables	127
3.2.10	gds.options.path_filename	130
3.3	Save your chip design	130
3.4	Routing Between QComponents	142
3.4.0.1	Introduction	142
3.4.0.2	Prerequisite	142
3.4.1	Example 1: Straight routing between two pins	143
3.4.2	Example 2: Any direction	144
3.4.3	Example 3: Angles and leads	144
3.4.4	Example 4: 90° angles	145
3.4.5	Example 5: Tight control on leads and angles	147
3.5	Simple Meander	149
3.5.0.1	Preparations	149
3.6	Using CPW meanders to connect the 3 Qbits	150
3.7	Hybrid Auto and AStar	158
3.7.1	Preparations	158
3.8	Get them all with MixedRoute	162
3.8.0.0.1	Single CPW using one meander and 3 simple segments	163
3.8.0.0.2	Single CPW using the pathfinder segments	164
3.9	Capacitance matrix and LOM analysis	168
3.9.0.1	Prerequisite	168
3.9.1	1. Create the design in Metal	168
3.9.1.1	In this example, the design consists of 4 qubits and 4 CPWs	168
3.9.2	2. Capacitance Analysis and LOM derivation using the analysis package - most users	170
3.9.2.1	Capacitance Analysis	170
3.9.2.2	Lumped oscillator model (LOM)	175
3.9.3	3. Directly access the renderer to modify other parameters	177
3.9.3.1	Code to swap rows and columns in capacitance matrix	178
3.10	Eigenmode and EPR analysis	180
3.10.0.1	Prerequisite	180
3.10.1	Sections	180
3.10.1.1	I. Transmon only	180
3.10.1.2	II. Resonator only	180
3.10.1.3	III. Transmon & resonator	180
3.10.1.4	IV. Analyze a coupled 2 transmon system.	180
3.11	1. Analyze the transmon qubit by itself	181
3.11.0.1	Create the Qbit design	181
3.11.0.2	Finite Element Eigenmode Analysis	182
3.11.0.2.1	Setup	182

3.11.0.2.2 Execute simulation and verify convergence and EM field	183
3.11.0.3 EPR Analysis	186
3.11.0.3.1 Setup	186
3.11.0.3.2 Mode frequencies (MHz)	191
3.11.0.3.3 Kerr Non-linear coefficient table (MHz)	191
3.12 2. Analyze the CPW resonator by itself	191
3.12.0.1 Update the design in Metal	191
3.12.0.2 Finite Element Eigenmode Analysis	191
3.12.0.2.1 Setup	191
3.12.0.2.2 Execute simulation and verify convergence and EM field	192
3.12.0.2.3 Refine	194
3.12.0.3 EPR Analysis	195
3.13 3. Analyze the combined transmon + CPW resonator system	196
3.13.0.1 Finite Element Eigenmode Analysis	196
3.13.0.1.1 Setup	196
3.13.0.1.2 Execute simulation and verify convergence and EM field	196
3.13.0.2 EPR Analysis	198
3.13.0.2.1 Mode frequencies (MHz)	203
3.13.0.2.2 Kerr Non-linear coefficient table (MHz)	204
3.14 4. Analyze a coupled 2-transmon system	204
3.14.0.1 Create the design	204
3.14.0.2 Finite Element Eigenmode Analysis	206
3.14.0.2.1 Setup	206
3.14.0.3 EPR Analysis	209
3.14.0.3.1 Setup	209
3.14.0.3.2 Mode frequencies (MHz)	215
3.14.0.3.3 Kerr Non-linear coefficient table (MHz)	215
3.14.0.4 1. load fluxonium cell Q3d simulation results	217
3.14.0.5 load transmon cell Q3d simulation results	217
3.14.0.6 2. Create LOM cells from capacitance matrices	217
3.14.0.6.1 Setting cell objects corresponding to the capacitance simulation results	217
3.14.0.7 3. Create subsystems	218
3.14.0.7.1 Creating the four subsystems, corresponding to the 2 qubits	218
3.14.0.8 4. Creat the composite system from the cells and the subsystems	219
3.14.0.9 5. Generate the hilberspace from the composite system, leveraging the scqubits package	220
3.14.0.10 6. Print the results	221
3.14.0.11 7. let's sweep some parameters now	222
3.14.0.11.1 Plot transition frequencies as a function of the flux	223

3.14.0.11.2	The dispersive shift, χ between the two qubits as a function of the flux	224
3.14.0.11.3	Zooming on the fluxonium sweet spot: its $0 \rightarrow 1$ transition as a function of the flux	225
3.14.0.11.4	Coherences of the fluxonium as a function of the flux	226
3.15	Analyzing and tuning a transmon qubit	231
3.15.1	Index	231
3.15.1.0.1	Transmon design	231
3.15.1.0.2	Transmon analysis using EPR method	231
3.15.1.0.3	Transmon analysis using LOM method	231
3.15.2	Prerequisite	231
3.16	1. Create the Qbit design	231
3.17	2. Analyze the transmon using the Eigenmode-EPR method	232
3.17.0.1	Finite Element Eigenmode Analysis	232
3.17.0.1.1	Setup	232
3.17.0.1.2	Execute simulation and verify convergence	233
3.17.0.1.3	Plot the EM field for inspection	235
3.17.0.2	EPR Analysis	236
3.17.0.2.1	Execute the energy distribution analysis	237
3.17.0.2.2	Run the EPR analysis	237
3.17.0.2.3	Mode frequencies (MHz)	241
3.17.0.2.4	Kerr Non-linear coefficient table (MHz)	241
3.18	3. Analyze the transmon using the LOM method	242
3.18.0.1	Capacitance matrix extraction	242
3.18.0.1.1	Setup	242
3.18.0.1.2	Execute simulation and verify convergence	243
3.18.0.2	LOM Analysis	244
3.19	Analyzing and tuning a resonator	247
3.19.0.1	Resonator design	247
3.19.0.2	Resonator analysis using EPR method	247
3.19.1	Prerequisite	247
3.20	1. Create the Resonator design	247
3.21	2. Analyze the resonator using the Eigenmode-EPR method	248
3.21.0.1	Finite Element Eigenmode Analysis	248
3.21.0.1.1	Setup	248
3.21.0.1.2	Execute simulation and verify convergence	249
3.21.0.1.3	Refine the resonator design, rerun simulation and verify convergence	251
3.21.0.1.4	Plot the EM field for inspection	253
3.21.0.2	EPR Analysis	254
3.22	Analyzing and tuning a transmon qubit with a resonator	257
3.22.1	Index	257
3.22.1.0.1	Transmon & resonator design	257
3.22.1.0.2	Transmon & resonator analysis using EPR method	257
3.22.1.0.3	Transmon & resonator analysis using LOM method	257

3.22.2 Prerequisite	257
3.23 1. Create the Qbit design	257
3.24 2. Analyze the transmon & resonator using the Eigenmode-EPR method	258
3.24.0.1 Finite Element Eigenmode Analysis	259
3.24.0.1.1 Setup	259
3.24.0.1.2 Execute simulation and verify convergence	260
3.24.0.1.3 Plot the EM field for inspection	262
3.24.0.2 EPR Analysis	264
3.24.0.2.1 Execute the energy distribution analysis	264
3.24.0.2.2 Run EPR analysis	265
3.24.0.2.3 Mode frequencies (MHz)	270
3.24.0.2.4 Kerr Non-linear coefficient table (MHz)	270
3.25 3. Analyze the transmon using the LOM method	270
3.25.0.1 Capacitance matrix extraction	270
3.25.0.1.1 Setup	270
3.25.0.1.2 Execute simulation and verify convergence	271
3.25.0.2 LOM Analysis	273
3.26 Analyzing a double hanger resonator (S Param)	276
3.26.0.1 Prerequisite	276
3.26.1 Create the design in Metal	276
3.26.2 2. Eigenmode and Impedance analysis using the advanced flow	279
3.26.2.0.1 Setup	279
3.26.2.0.2 Execute simulation and verify convergence	280
3.26.2.0.3 Execute simulation and observe the Impedance	280
3.27 Example full chip design	284
3.28 Layout	284
3.28.0.0.1 The Qubits	285
3.28.0.0.2 The Busses	290
3.28.0.0.3 The Readouts and Control Lines	294
3.29 Analyze	304
3.29.1 Capacitance Extraction and LOM	304
3.29.2 Eigenmode and EPR	314
3.29.2.0.1 Preparations	314
3.29.2.0.2 Mode frequencies (MHz)	323
3.29.2.0.3 Kerr Non-linear coefficient table (MHz)	323
3.29.2.1 Rendering to a GDS File	323

Chapter 1

Introduction

Quantum processors harness the intrinsic properties of quantum mechanical systems – such as quantum parallelism and quantum interference – to solve certain problems where classical computers fall short.

One prominent platform for constructing a multi-qubit quantum processor involves superconducting qubits, in which information is stored in the quantum degrees of freedom of nanofabricated, anharmonic oscillators constructed from superconducting circuit elements. In contrast to other platforms, e.g. electron spins in silicon and quantum dots, trapped ions, ultracold atoms, nitrogen-vacancies in diamonds, and polarized photons, where the quantum information is encoded in natural microscopic quantum systems, superconducting qubits are macroscopic in size and litho-graphically defined.

One remarkable feature of superconducting qubits is that their energy-level spectra are governed by circuit element parameters and thus are configurable; they can be designed to exhibit “atom-like” energy spectra with desired properties. Therefore, superconducting qubits are also often referred to as artificial atoms, offering a rich parameter space of possible qubit properties and operation regimes, with predictable performance in terms of transition frequencies, anharmonicity, and complexity.

1.1 Introduction To Electronics

Electronics is a branch of physics and electrical engineering that deals with the emission, behaviour and effects of electrons using electronic devices. This chapter will cover the bare minimum electronics information needed to understand quantum processor theory and design.

1.1.1 Capacitors

The capacitor is a component which has the ability or “capacity” to store energy in the form of an electrical charge producing a potential difference across its plates, much like a small rechargeable battery.

In its basic form, a capacitor consists of two or more parallel conductive plates which are not connected or touching each other, but are electrically separated either by air or by some form of a good insulating material such as waxed paper, mica, ceramic, plastic or some form of a liquid gel as used in electrolytic capacitors. The insulating layer between a capacitor's plates is commonly called the dielectric.

Due to this insulating layer, DC current can not flow through the capacitor as it blocks it allowing instead a voltage to be present across the plates in the form of an electrical charge.

The energy within the charge is stored in an “electrostatic field” between the two plates. When an electric current flows into the capacitor, it charges up, so the electrostatic field becomes much stronger as it stores more energy between the plates.

Likewise, as the current flowing out of the capacitor, discharging it, the potential difference between the two plates decreases and the electrostatic field decreases as the energy moves out of the plates.

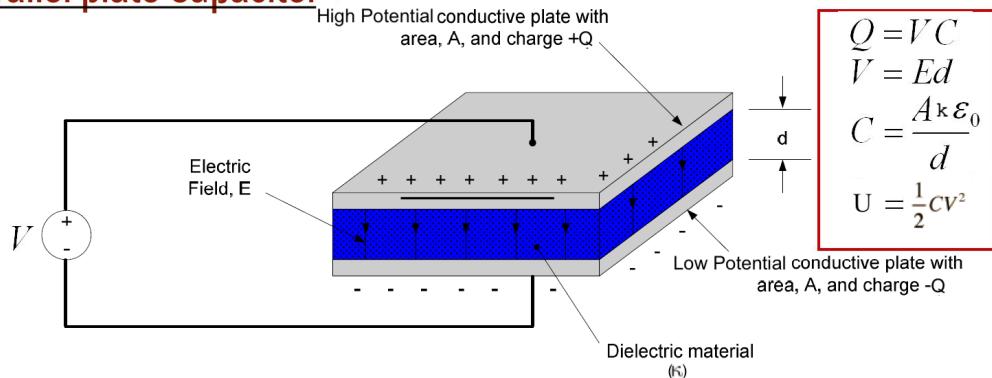
The property of a capacitor to store charge on its plates in the form of an electrostatic field is called the Capacitance of the capacitor. Not only that, but capacitance is also the property of a capacitor which resists the change of voltage across it.

1.1.2 Inductors

An inductor, also called a choke, is another passive type electrical component consisting of a coil of wire designed to take advantage of this relationship by inducing a magnetic field in itself or within its core as a result of the current flowing through the wire coil. Forming a wire coil into an inductor results in a much stronger magnetic field than one that would be produced by a simple coil of wire.

The current, i that flows through an inductor produces a magnetic flux that is pro-

Parallel-plate Capacitor

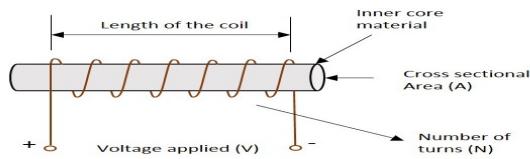


portional to it. But unlike a Capacitor which oppose a change of voltage across their plates, an inductor opposes the rate of change of current flowing through it due to the build up of self-induced energy within its magnetic field.

In other words, inductors resist or oppose changes of current but will easily pass a steady state DC current. This ability of an inductor to resist changes in current and which also relates current, i with its magnetic flux linkage, $N\Phi$ as a constant of proportionality is called Inductance which is given the symbol L with units of Henry, (H) after Joseph Henry.

A time varying magnetic field induces a voltage that is proportional to the rate of change of the current producing it with a positive value indicating an increase in emf and a negative value indicating a decrease in emf. The equation relating this self-induced voltage, current and inductance can be found by substituting the $\mu N^2 A/l$ with L denoting the constant of proportionality called the Inductance of the coil.

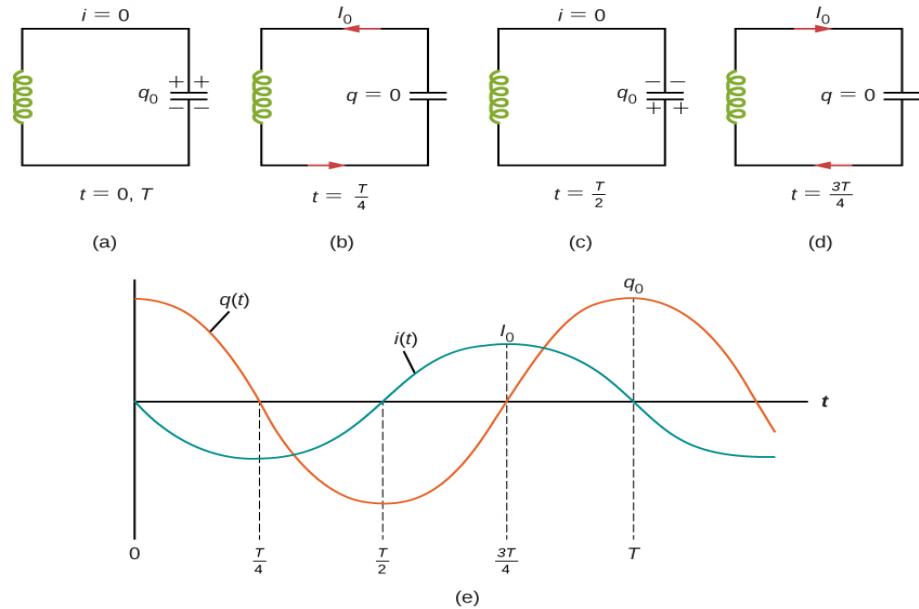
The relation between the flux in the inductor and the current flowing through the inductor is given as: $N\Phi = Li$. As an inductor consists of a coil of conducting wire, this then reduces the above equation to give the self-induced emf, sometimes called the back emf induced in the coil too.



1.1.3 LC Circuits

It is worth noting that both capacitors and inductors store energy, in their electric and magnetic fields, respectively. A circuit containing both an inductor (L) and a capacitor

(C) can oscillate without a source of emf by shifting the energy stored in the circuit between the electric and magnetic fields. Thus, the concepts we develop in this section are directly applicable to the exchange of energy between the electric and magnetic fields in electromagnetic waves, or light. We start with an idealized circuit of zero resistance that contains an inductor and a capacitor, an *LC* circuit.



An LC circuit is shown in figure 1.3. If the capacitor contains a charge q_0 before the switch is closed, then all the energy of the circuit is initially stored in the electric field of the capacitor. This energy is

$$U_c = \frac{1}{2} \frac{q_0^2}{C} \quad (1.1)$$

When the switch is closed, the capacitor begins to discharge, producing a current in the circuit. The current, in turn, creates a magnetic field in the inductor. The net effect of this process is a transfer of energy from the capacitor, with its diminishing electric field, to the inductor, with its increasing magnetic field.

In figure 1.3(b), the capacitor is completely discharged and all the energy is stored in the magnetic field of the inductor. At this instant, the current is at its maximum value I_0 and the energy in the inductor is

$$U_L = \frac{1}{2} L I_0^2 \quad (1.2)$$

Since there is no resistance in the circuit, no energy is lost through Joule heating; thus, the maximum energy stored in the capacitor is equal to the maximum energy stored at a later time in the inductor:

$$\frac{1}{2} \frac{q_0^2}{C} = \frac{1}{2} L I_0^2 \quad (1.3)$$

At an arbitrary time when the capacitor charge is $q(t)$ and the current is $i(t)$, the total energy U in the circuit is given by

$$U = \frac{1}{2} \frac{q^2}{C} + \frac{1}{2} L I^2 = \frac{1}{2} \frac{q_0^2}{C} = \frac{1}{2} L I_0^2 \quad (1.4)$$

After reaching its maximum I_0 , the current $i(t)$ continues to transport charge between the capacitor plates, thereby recharging the capacitor. Since the inductor resists a change in current, current continues to flow, even though the capacitor is discharged. This continued current causes the capacitor to charge with opposite polarity. The electric field of the capacitor increases while the magnetic field of the inductor diminishes, and the overall effect is a transfer of energy from the inductor back to the capacitor. From the law of energy conservation, the maximum charge that the capacitor re-acquires is q_0 . However, as figure 1.3(c) shows, the capacitor plates are charged opposite to what they were initially.

When fully charged, the capacitor once again transfers its energy to the inductor until it is again completely discharged, as shown in figure 1.3(d). Then, in the last part of this cyclic process, energy flows back to the capacitor, and the initial state of the circuit is restored.

We have followed the circuit through one complete cycle. Its electromagnetic oscillations are analogous to the mechanical oscillations of a mass at the end of a spring. In this latter case, energy is transferred back and forth between the mass, which has kinetic energy $mv^2/2$, and the spring, which has potential energy $kx^2/2$. With the absence of friction in the mass-spring system, the oscillations would continue indefinitely. Similarly, the oscillations of an LC circuit with no resistance would continue forever if undisturbed; however, this ideal zero-resistance LC circuit is not practical, and any LC circuit will have at least a small resistance, which will radiate and lose energy over time.

1.2 Introduction To Quantum Mechanics

1.2.1 Quantum Harmonic Oscillator

The quantum harmonic oscillator is the quantum-mechanical analog of the classical harmonic oscillator. Since an arbitrary smooth potential can usually be approximated as a harmonic potential at the vicinity of a stable equilibrium point, it is one of the most important model systems in quantum mechanics. Furthermore, it is one of the few quantum-mechanical systems for which an exact, analytical solution is known.

For a one dimensional quantum harmonic oscillator, the Hamiltonian of a particle can be written as

$$\hat{H} = \frac{\hat{p}^2}{2m} + \frac{1}{2}k\hat{x}^2 = \frac{\hat{p}^2}{2m} + \frac{1}{2}m\omega^2\hat{x}^2 \quad (1.5)$$

where m is the particle's mass, k is the force constant, $\omega = \sqrt{k/m}$ is the angular frequency of the oscillator, \hat{x} is the position operator and \hat{p} is the momentum operator. The first term in the Hamiltonian represents the kinetic energy of the particle, and the second term represents its potential energy, as in Hooke's Law.

Then the Schrödinger Equation can be written as

$$\hat{H} |\psi\rangle = E |\psi\rangle \quad (1.6)$$

where E denotes a to-be-determined real number that will specify a time-dependent energy level, or eigenvalue, and the solution $|\psi\rangle$ denotes that level's energy eigenstate.

When the differential equation is solved, the solution turns out to be the Hermite functions,

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega x^2}{2\hbar}} H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right) \quad (1.7)$$

where the corresponding energy levels are

$$E_n = \hbar\omega(n + \frac{1}{2}) = (2n + 1)\frac{\hbar}{2}\omega. \quad (1.8)$$

This energy spectrum is noteworthy for three reasons. First, the energies are quantized, meaning that only discrete energy values (integer-plus-half multiples of $\hbar\omega$) are possible; this is a general feature of quantum-mechanical systems when a particle is confined. Second, these discrete energy levels are equally spaced, unlike in the Bohr model of the atom, or the particle in a box. Third, the lowest achievable energy (the energy of the $n = 0$ state, called the ground state) is not equal to the minimum of the potential well, but $\hbar\omega/2$ above it; this is called zero-point energy. Because of the zero-point energy, the position and momentum of the oscillator in the ground state are not fixed (as they would be in a classical oscillator), but have a small range of variance, in accordance with the Heisenberg uncertainty principle.

1.2.2 Lamb Shift

According to the hydrogen Schrödinger equationsolution, the energy levels of the hydrogen electron should depend only on the principal quantum number n . In 1951, Willis Lamb discovered that this was not so - that the $2p(1/2)$ state is slightly lower than the $2s(1/2)$ state resulting in a slight shift of the corresponding spectral line.

At the heart of this process is the exchange force by which chargers interact by the exchange of photons. There can be a self-interaction of the electron by exchange of a photo which "smears out" the electron position over a range of about 0.1 fermi. This causes the electron spin to be slightly different from 2. There is also a slight weakening of the force on the electron when it is very close to the nucleus, causing the $2s$ electron (which has penetration all the way to the nucleus) to be slightly higher in energy than the $2p(1/2)$ electron.

1.2.3 Kerr Effect

The Kerr electro-optic effect, or DC Kerr effect, is the special case in which slowly varying external electric field is applied by, for instance, a voltage on electrodes across the sample material. Under this influence, the sample becomes birefringent, whith different indices of refraction for light polarized parallel to or perpendicular to the applied field. The difference in index of refraction, Δn , is given by

$$\Delta n = \lambda K E^2 \quad (1.9)$$

where λ is the wavelength of the light, K is the Kerr constant, and E is the strength of the electric field. This difference in index of refraction causes the material to act like a waveplate when light is incident on it in a direction perpendicular to the electric field. If the material is placed between two "crossed" (perpendicular) linear polarizers, no light will be transmitted when the electric field is turned off, while nearly all of the light will be transmitted for some optimum value of the electric field. Higher values of the Kerr constant allow complete transmission to be achieved with a smaller applied electric field.

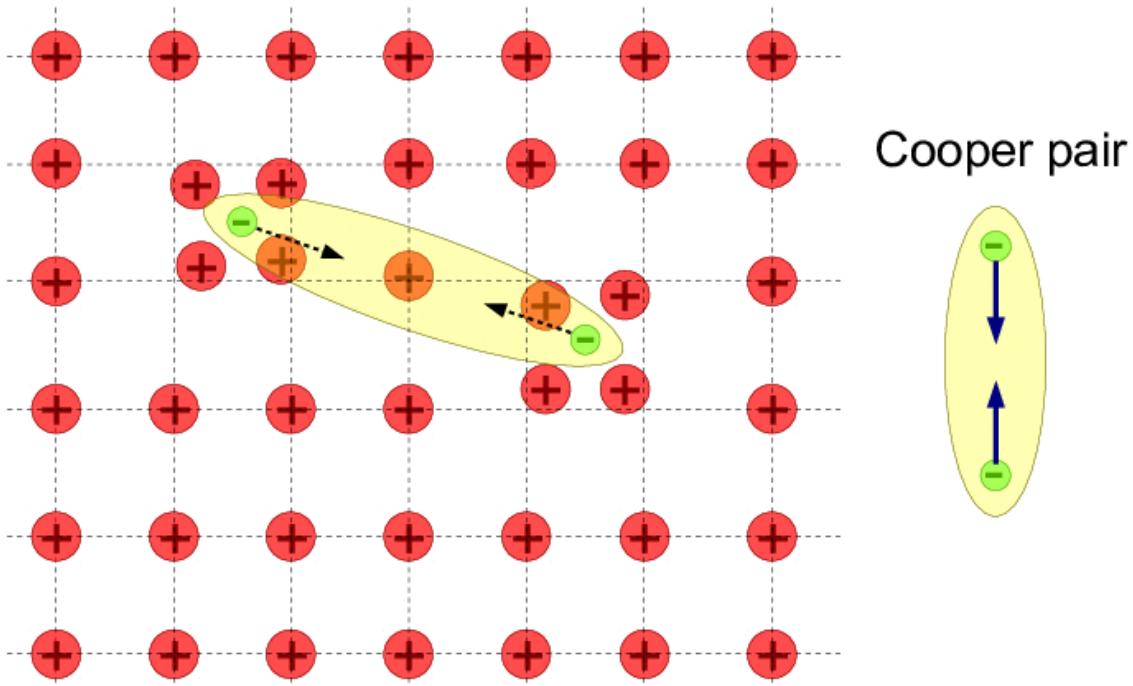
1.2.4 Tunneling

Quantum tunneling is a quantum mechanical phenomenon whereby a wavefunction can propagate through a potential barrier. he transmission through the barrier can be finite and depends exponentially on the barrier height and barrier width. The wavefunction may disappear on one side and reappear on the other side. The wavefunction and its first derivative are continuous. In steady-state, the probability flux in the forward direction is spatially uniform. No particle or wave is lost.

Using mathematical formulations, such as the Schrödinger equation, the wave function can be deduced. The square of the absolute value of this wavefunction is directly related to the probability distribution of the particle's position, which describes the probability that the particle is at any given place. The wider the barrier and the higher the barrier energy, the lower the probability of tunneling.

1.2.5 Cooper Pairs (Bardeen- Cooper- Schrieffer pair)

Bardeen-Cooper-Schrieffer Theory is a formulation of superconductivity which explains phenomenon in which a current of electron pairs flows without resistance in certain materials at low temperatures. Normally an electron would never be able to interact with another electron due to the repulsive effect of the Coulomb force. When a single negatively charged electron slightly distorts the lattice of atoms in the superconductor, drawing toward it a small excess of positive charge, this excess attracts a second electron. In most materials this attractive interaction comes from phonon exchange. To create a superconducting metal, there should be sufficiently strong electron phonon interaction. Basically, one electron emits a phonon and changes direction and another one absorbs the phonon and changes its own direction. This phonon exchange process creates a correlated state between two electrons. This pair of electrons (or other fermions) called as Copper pairs or BCS pairs.



Chapter 2

Circuit Quantum Electrodynamics

Circuit quantum electrodynamics (QED) is the study of the interaction of nonlinear superconducting circuits, acting as artificial atoms or as qubits for quantum information processing, with quantized electromagnetic fields in the microwave frequency domain.

Circuit QED combines the theoretical and experimental tools of atomic physics, quantum optics and the physics of mesoscopic superconducting circuits not only to further explore the physics of cavity QED and quantum optics in novel parameter regimes, but also to allow the realization of engineered quantum devices with technological applications.

2.1 Superconducting Quantum Circuits

Circuit components with spatial dimensions that are small compared to the relevant wavelength can be treated as lumped elements, and we start this section with a particularly simple lumped-element circuit: the quantum LC oscillator. We subsequently discuss the closely related two- and three-dimensional microwave resonators that play a central role in circuit QED experiments and which can be thought of as distributed versions of the LC oscillator with a set of harmonic frequencies. Finally, we move on to nonlinear quantum circuits with Josephson junctions as the source of nonlinearity, and discuss how such circuits can behave as artificial atoms. We put special emphasis on the transmon qubit, which is the most widely used artificial atom design in current circuit QED experiments.

2.1.1 The Quantum LC Resonator

An *LC* oscillator is characterized by its inductance L and capacitance C or, equivalently, by its angular frequency $\omega_r = \frac{1}{\sqrt{LC}}$ and characteristic impedance $Z_r = \sqrt{\frac{L}{C}}$. The total energy of this oscillator is given by the sum of its charging and inductive energy

$$H_{LC} = \frac{Q^2}{2C} + \frac{\Phi^2}{2L} \quad (2.1)$$

where Q is the charge on the capacitor and Φ the flux threading the inductor.

It is instructive to rewrite H_{LC} as

$$H_{LC} = \frac{Q^2}{2C} + \frac{1}{2}C\omega_r^2\Phi^2 \quad (2.2)$$

This form emphasizes the analogy of the *LC* oscillator with a mechanical oscillator of coordinate Φ , conjugate momentum Q , and mass C . With this analogy in mind, quantization proceeds in a manner that should be well known to the reader: The charge and flux variables are promoted to non-commuting observables satisfying the commutation relation

$$[\hat{\Phi}, \hat{Q}] = i\hbar \quad (2.3)$$

It is further useful to introduce the standard annihilation \hat{a} and creation \hat{a}^\dagger operators of the harmonic oscillator. Since $\hat{x} = \sqrt{\frac{\hbar}{2m\omega}}(\hat{a}^\dagger + \hat{a})$, $\hat{p} = i\sqrt{\frac{\hbar m\omega}{2}}(\hat{a}^\dagger - \hat{a})$ and with the above mechanical analogy in mind, we choose these operators as

$$\hat{\Phi} = \Phi_{zpf}(\hat{a}^\dagger + \hat{a})\hat{Q} = iQ_{zpf}(\hat{a}^\dagger - \hat{a}) \quad (2.4)$$

with $\Phi_{zpf} = \sqrt{\frac{\hbar}{2\omega_r C}} = \sqrt{\frac{\hbar Z_r}{2}}$ and $Q_{zpf} = \sqrt{\frac{\hbar\omega_r C}{2}} = \sqrt{\frac{\hbar}{2Z_r}}$ the characteristic magnitude of the zero-point fluctuations of the flux and the charge, respectively. With these definitions, the above Hamiltonian takes the usual form

$$\hat{H}_{LC} = \hbar\omega_r(\hat{a}^\dagger\hat{a} + \frac{1}{2}) \quad (2.5)$$

with eigenstates that satisfy $\hat{a}^\dagger\hat{a}|n\rangle = n|n\rangle$ for $n = 0, 1, 2, \dots$. For the rest of the calculations, we follow the convention of dropping from the Hamiltonian the factor of $\frac{1}{2}$ corresponding to zero-point energy. The action of $\hat{a}^\dagger = \sqrt{\frac{1}{2\hbar Z_r}}(\hat{\Phi} - iZ_r\hat{Q})$ is to create a quantized excitation of the flux and charge degrees of freedom of the oscillator or, equivalently of the magnetic and electric fields. **In other words, \hat{a}^\dagger creates a photon of frequency ω_r stored in the circuit.**

While formally correct, one can wonder if this quantization procedure is relevant in practice. In other words, is it possible to operate *LC* oscillators in a regime where quantum effects are important? For this to be the case, at least two conditions must be satisfied. First, the oscillator should be sufficiently well decoupled from uncontrolled degrees of freedom such that its energy levels are considerably less broad than their separation. In other words, we require the oscillator's quality factor $Q = \omega_r/\kappa$, with κ the oscillator linewidth or equivalently the photon loss rate, to be large. Because losses are at the origin of level broadening, superconductors are ideal to reach the quantum regime. In practice, most circuit QED devices are made of thin aluminum films evaporated on low-loss dielectric substrates such as sapphire or high-resistivity silicon wafers. Mainly for its larger superconducting gap, niobium is sometimes used in place of aluminum. In addition to internal losses in the metals forming the LC circuit, care must also be taken to minimize the effect of coupling to the external circuitry that is essential for operating the oscillator. As will be discussed below, large quality factors ranging from $Q \approx 10^3$ to 10^8 can be obtained in the laboratory.

With these two requirements satisfied, an oscillator with a frequency in the microwave range can be operated in the quantum regime. This means that the circuit can be prepared in its quantum-mechanical ground state $|n=0\rangle$ simply by waiting for a time of the order of a few photon lifetimes $T_\kappa = \frac{1}{\kappa}$. It is also crucial to note that the vacuum fluctuations in such an oscillator can be made large. For example, taking reasonable values $L \approx 0.8$ nH and $C \approx 0.4$ pF, corresponding to $\frac{\omega_r}{2\pi} \approx 8$ GHz and $Z_r \approx 50\Omega$, the ground state is characterized by vacuum fluctuations of the voltage of variance as large as $\delta V_0 = [\langle \hat{V}^2 \rangle - \langle \hat{V} \rangle^2]^{1/2} = \sqrt{\frac{\hbar\omega_r}{2C}} \approx 1\mu V$, with $\hat{V} = \frac{\hat{Q}}{C}$. As will be made clear later, this leads to large electric field fluctuations and therefore to large electric dipole interactions when coupling to an artificial atom.

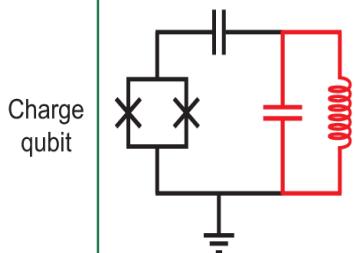
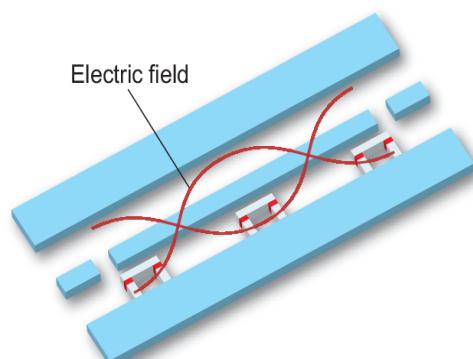
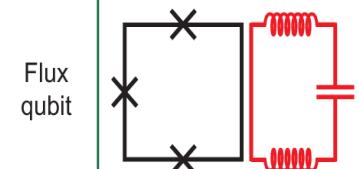
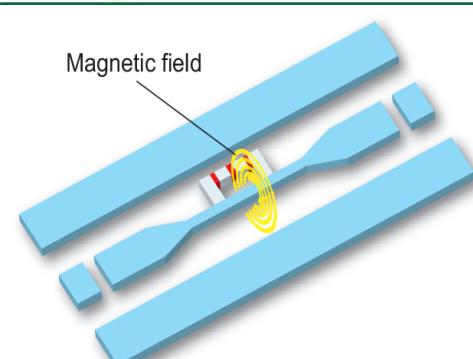
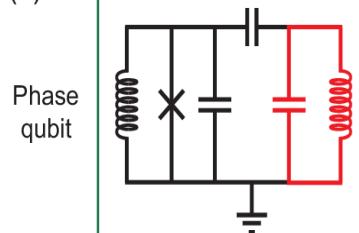
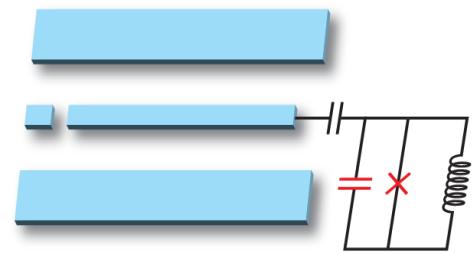
2.1.2 2D Resonators

Quantum harmonic oscillators come in many shapes and sizes, the LC oscillator being just one example. Other types of harmonic oscillators that feature centrally in circuit QED are microwave resonators where the electromagnetic field is confined either in a planar, essentially two-dimensional structure (2D resonators) or in a three-dimensional volume (3D resonators). The boundary conditions imposed by the geometry of these different resonators lead to a discretization of the electromagnetic field into a set of modes with distinct frequencies, where each mode can be thought of as an independent harmonic oscillator. Conversely (especially for the 2D case) one can think of these modes as nearly dissipationless acoustic plasma modes of superconductors.

A coplanar waveguide resonator consist of a coplanar waveguide of finite length formed by a center conductor of width w and thickness t , separated on both sides by a distance s from a ground plane of the same thickness. Both conductors are typically deposited on a low-loss dielectric substrate of permittivity ϵ and thickness much larger than the dimensions w , s , t . This planar structure acts as a transmission line along which signals are transmitted in a way analogous to a conventional coaxial cable. As in a coaxial cable, the coplanar waveguide confines the electromagnetic field to a small volume between its center conductor and the ground. The dimensions of the center conductor, the gaps, and the thickness of the dielectric are chosen such that the field is concentrated between the center conductor and ground, and radiation in other directions is minimized. This structure supports a quasi-TEM mode, with the electromagnetic field partly in the dielectric substrate and in the vacuum (or other dielectric) above the substrate, and with the largest concentration in the gaps between the center conductor and the ground planes. In practice, the coplanar waveguide can be treated as an essentially dispersion-free, linear dielectric medium. Ideally, the loss of coplanar waveguides is only limited by the conductivity of the center conductor and the ground plane, and by the loss tangent of the dielectric. To minimize losses, superconducting metals such as aluminum, niobium or niobium titanium nitride (NbTiN), are used in combination with dielectrics of low loss tangent, such as sapphire or high-resistivity silicon.

Similarly to the lumped LC oscillator, the electromagnetic properties of a coplanar waveguide resonator are described by its characteristic impedance $Z_r = \sqrt{\frac{l_0}{c_0}}$ and the speed of light in the waveguide $v_0 = \frac{1}{\sqrt{l_0 c_0}}$, where we have introduced the capacitance to ground c_0 and inductance l_0 per unit length. Typical values of these parameters are $Z_r \approx 50\Omega$ and $v_0 \approx 1.3 \times 10^8$ m/s, or about a third of the speed of light in vacuum. For a given substrate, metal thickness and center conductor width, the characteristic impedance can be adjusted by varying the parameters of the waveguide.

A resonator is formed from a coplanar waveguide by imposing boundary conditions of either zero current or zero voltage at the two endpoints separated by a distance. Zero

	LC oscillator	Coplanar waveguide resonator
(a)	<p>Charge qubit</p> 	 <p>Electric field</p>
(b)	<p>Flux qubit</p> 	 <p>Magnetic field</p>
(c)	<p>Phase qubit</p> 	

current is achieved by micro-fabricating a gap in the center conductor (open boundary), while zero voltage can be achieved by grounding an end point (shorted boundary).

A typical example is a $\lambda/2$ resonator of length 1.0 cm and speed of light 1.3×10^8 m/s corresponding to a fundamental frequency of 6.5 GHz. This coplanar waveguide geometry is very flexible and a large range of frequencies can be achieved. In practice, however, the useful frequency range is restricted from above by the superconducting gap of the metal from which the resonator is made (82 GHz for aluminum). Above this energy, losses due to quasiparticles increase dramatically. Low frequency resonators can be made by using long, meandering, coplanar waveguides. For example, a resonator was realized with a length of 0.68 m and a fundamental frequency of $f_0 = 92$ MHz. With this frequency corresponding to a temperature of 4.4 mK, the low frequency modes of such long resonators are, however, not in the vacuum state. Indeed, according to the Bose-Einstein distribution, the thermal occupation of the fundamental mode frequency at 10 mK is $n_{\kappa} = \frac{1}{e^{hf_0/k_B T} - 1} \approx 1.8$. Typical circuit QED experiments rather work with resonators in the range of 5 – 15 GHz where, conveniently, microwave electronics is well developed.

As already mentioned, entering the quantum regime for a given mode m requires more than $\hbar\omega_m \gg k_B T$. It is also important that the linewidth κ_m be small compared to the mode frequency ω_m . As for the LC oscillator, the linewidth can be expressed in terms of the quality factor Q_m of the resonator mode as $\kappa_m = \frac{\omega_m}{Q_m}$. There are multiple sources of losses and it is common to distinguish between internal losses due to coupling to uncontrolled degrees of freedom (dielectric and conductor losses at the surfaces and interfaces, substrate dielectric losses, non-equilibrium quasiparticles, vortices, two-level fluctuators...) and external losses due to coupling to the input and output ports used to couple signals in and out of the resonator. In terms of these two contributions, the total dissipation rate of mode m is $\kappa_m = \kappa_{ext,m} + \kappa_{int,m}$ and the total, or loaded, quality factor of the resonator is therefore $Q_{L,m} = (Q_{ext,m}^{-1} + Q_{int,m}^{-1})^{-1}$. It is always advantageous to maximize the internal quality factor and much effort have been invested in improving resonator fabrication such that values of $Q_{int} \approx 10^5$ are routinely achieved.

On the other hand, the external quality factor can be adjusted by designing the capacitive coupling at the open ends of the resonator to input/output transmission lines. In coplanar waveguide resonators, these input and output coupling capacitors are frequently chosen either as a simple gap of a defined width in the center conductor. The choice $Q_{ext} \ll Q_{int}$ corresponding to an 'overcoupled' resonator is deal for fast qubit measurement. On the other hand, undercoupled resonators, $Q_{ext} \gg Q_{int}$, where dissipation is only limited by internal losses which are kept as small as possible, can serve as quantum memories to store microwave photons for long times. Using different modes of the same resonator, or combinations of resonators, both regimes of high and low external losses can also be combined in the same circuit QED device.

Finally, the magnitude of the vacuum fluctuations of the electric field in coplanar

waveguide resonators is related to the mode volume. While the longitudinal dimension of the mode is limited by the length of the resonator, which also sets the fundamental frequency $d \approx \frac{\lambda}{2}$, the transverse dimension can be adjusted over a broad range. Commonly chosen transverse dimensions are on the order of $\omega \approx 10\mu\text{m}$ and $s \approx 5\mu\text{m}$. If desired, the transverse dimension of the center conductor may be reduced to the sub-micron scale, up to a limit set by the penetration depth of the superconducting thin films which is typically of the order of 100 to 200nm. When combining the typical separation $s \approx 5\mu\text{m}$ with the magnitude of the voltage fluctuations $\Delta V_0 \approx 1\mu\text{V}$ already expected from the discussion of the LC circuit, we find that the zero-point electric field in coplanar resonator can be as large as $\Delta E_0 = \frac{\Delta V_0}{s} \approx 0.2\text{V/m}$. This is at least two orders of magnitude larger than the typical amplitude of ΔE_0 in the 3D cavities used in cavity QED.

2.1.2.1 Quantized Modes Of Transmission Line Resonator

While only a single mode of the transmission line resonator is often considered, there are many circuit QED experiments where the multimode structure of the device plays an important role.

2.1.3 The Transmon Artificial Atom

Although the oscillators discussed in the previous section can be prepared in their quantum mechanical ground state, it is challenging to observe clear quantum behavior with such linear systems. Indeed, harmonic oscillators are always in the correspondence limit and some degree of non-linearity is therefore essential to encode and manipulate quantum information in these systems. Fortunately, superconductivity allows to introduce nonlinearity in quantum electrical circuits while avoiding losses. Indeed, the Josephson junction is a nonlinear circuit element that is compatible with the requirements for very high quality factors and operation at millikelvin temperatures.

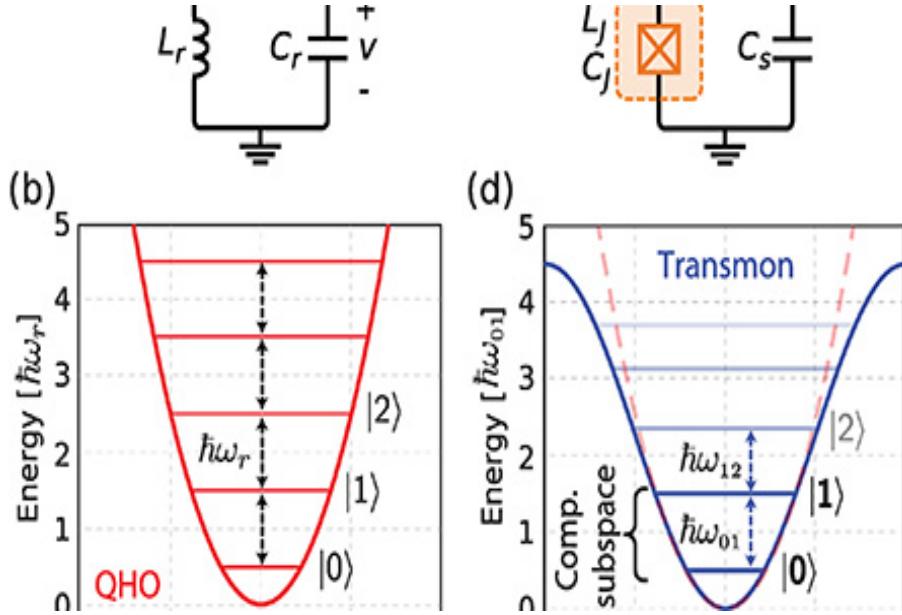
Contrary to expectations, Josephson showed that a dissipationless current, i.e. a supercurrent, could flow between two superconducting electrodes separated by a thin insulating barrier. More precisely, he showed that this supercurrent is given by $I = I_c \sin \phi$, where I_c is the junction's critical current and ϕ the phase difference between the superconducting condensates on either sides of the junction. The critical current, whose magnitude is determined by the junction size and material parameters, is the maximum current that can be supported before Cooper pairs are broken. Once this happens, dissipation kicks in and a finite voltage develops across the junction accompanied by a resistive current. Clearly, operation in the quantum regime requires currents well below this critical current. Josephson also showed that the time dependence of the phase difference ϕ is related to the voltage across the junction according to $\frac{d\phi}{dt} = \frac{2\pi V}{\Phi_0}$, with $\Phi_0 = \frac{\hbar}{2e}$ the flux quantum. It is useful to write

this expression as $\phi(t) = \frac{2\pi\Phi(t)}{\Phi_0} \pmod{2\pi} = 2\pi \int dt' V(t')/\Phi_0 \pmod{2\pi}$, with $\Phi(t)$ the flux variable already introduced. The mod 2π in the previous equalities takes into account the fact that the superconducting phase ϕ is a compact variable on the unit circle, $\phi = \phi + 2\pi$, while Φ can take arbitrary real values.

Taken together, the two Josephson relations make it clear that a Josephson junction relates current I to flux Φ . This is akin to a geometric inductance whose constitutive relation $\Phi = LI$ also links these two quantities. For this reason, it is useful to define the Josephson inductance as

$$L_J(\Phi) = \left(\frac{\partial I}{\partial \Phi} \right)^{-1} = \frac{\Phi_0}{2\pi I_c} \frac{1}{\cos(2\pi\Phi/\Phi_0)} \quad (2.6)$$

In contrast to geometric inductances, L_J depends on the flux. As a result, when operated below the critical current, the Josephson junction can be thought of as a nonlinear oscillator.



Replacing the geometric inductance L of the LC circuit oscillator discussed earlier by a Josephson junction renders the circuit nonlinear. In this situation, the energy levels of the circuit are no longer equidistant. If the nonlinearity and the quality factor of the junction are large enough, the energy spectrum resembles that of an atom, with well-resolved and nonuniformly spread spectral lines. We therefore often refer to this circuit as an **artificial atom**. In many situations, we can furthermore restrict our attention to only two energy levels, typically the ground and first excited states, forming a qubit.

To make this discussion more precise, it is useful to see how the Hamiltonian of the circuit is modified by the presence of the Josephson junction taking the place of the linear inductor. While the energy stored in a linear inductor is $E = \int dt V(t)I(t) = \int dt(d\Phi/dt)I = \Phi^2/2L$,

where we have used $\Phi = LI$ in the last equality, the energy of the nonlinear inductance rather takes the form

$$E = I_c \int dt \left(\frac{d\Phi}{dt} \right) \sin\left(\frac{2\pi\Phi}{\Phi_0}\right) == E - J \cos\left(\frac{2\pi\Phi}{\Phi_0}\right) \quad (2.7)$$

with $E = \frac{\Phi_0 I_c}{2\pi}$ the Josephson energy. This quantity is proportional to the rate of tunneling of Cooper pairs across the junction. Taking this contribution into account, the quantized Hamiltonian of the capacitively shunted Josephson junction therefore reads

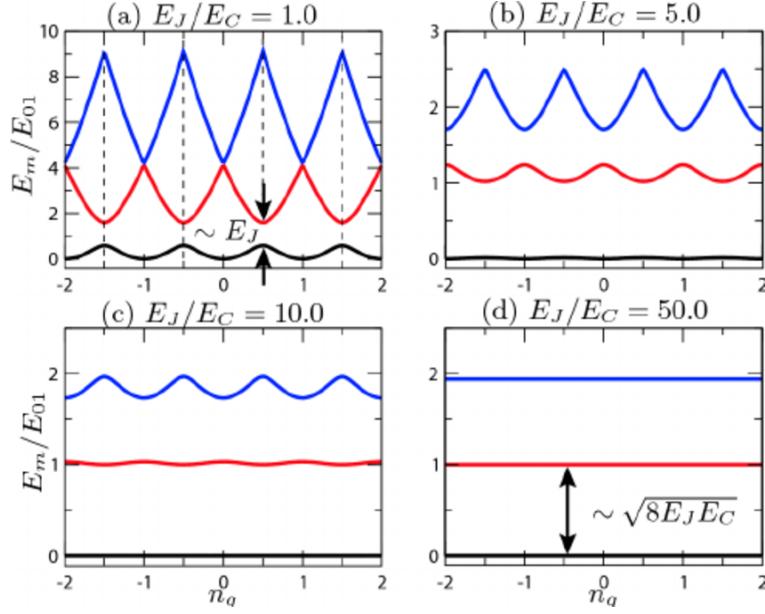
$$\hat{H}_T = \frac{(\hat{Q} - Q_g)^2}{2C\Sigma} - E_J \cos\left(\frac{2\pi\hat{\Phi}}{\Phi_0}\right) = 4E_C(\hat{n} - n_g)^2 - E_J \cos\hat{\phi} \quad (2.8)$$

In this expression, $C_\Sigma = C_J + C_S$ is the total capacitance, including the junction's capacitance C_J and the shunt capacitance C_S . In the second line, we have defined the charge number operator $\hat{n} = \hat{Q}/2e$, the phase operator $\hat{\phi} = (2\pi/\Phi_0)\hat{\Phi} \pmod{2\pi}$ and the charging energy $E_C = e^2/2C_\Sigma$. We have also included a possible offset charge $n_g = Q_g/2e$ due to capacitive coupling of the transmon to external charges. The offset charge can arise from spurious unwanted degrees of freedom in the transmon's environment or from an external gate voltage $V_g = Q_g/C_g$. As we show below, the choice of E_J and E_C is crucial in determining the system's sensitivity to the offset charge.

The spectrum of \hat{H}_T is controlled by the ratio E_J/E_C , with different values of this ratio corresponding to different types of superconducting qubits. Regardless of the parameter regime, one can always express the Hamiltonian in the diagonal form $\hat{H} = \sum_j \hbar\omega_j |j\rangle\langle j|$

in terms of its eigenfrequencies ω_j and eigenstates $|j\rangle$.

Figure 2.3 shows the energy difference $\omega_j - \omega_0$ for the three lowest energy levels for different ratios E_J/E_C as obtained from numerical diagonalization of \hat{H}_T . If the charging energy dominates, $E_J/E_C < 1$, the eigenstates of the Hamiltonian are approximately given by eigenstates of the charge operator, $|j\rangle \approx |n\rangle$, with $\hat{n}|n\rangle = n|n\rangle$. In this situation, a change in gate charge n_g has a large impact on the transition frequency of the device. As a result, unavoidable charge fluctuations in the circuit's environment lead to corresponding fluctuations in the qubit transition frequency and constantly dephasing.



To mitigate this problem, a solution is to work in the **transmon regime** where the ratio E_J/E_C is large. Typical values are $E_J/E_C \approx 20 - 80$. In this situation, the charge degree of freedom is highly delocalized due to the large Josephson energy. For this reason, the first energy levels of the device become essentially independent of the gate charge. It can in fact be shown that the charge dispersion, which describes the variation of the energy levels with gate charge, decreases exponentially with E_J/E_C . However, it is also clear that the price to pay for this increased coherence is the reduced anharmonicity of the transmon, anharmonicity that is required to control the qubit without causing unwanted transitions to higher excited states. Fortunately, while charge dispersion is exponentially small with E_J/E_C , the loss of anharmonicity has a much weaker dependance on this ratio given by $\approx (E_J/E_C)^{-1/2}$. The reduction in anharmonicity is not an impediment to controlling the transmon state with high fidelity.

While the variance of the charge degree of freedom is large when $E_J/E_C \gg 1$, the variance of its conjugate variable $\hat{\phi}$ is correspondingly small, with $\Delta\hat{\phi} = \sqrt{\langle\hat{\phi}^2\rangle - \langle\hat{\phi}\rangle^2} \ll 1$. In this situation, it is instructive to write the Hamiltonian as

$$\hat{H}_T = 4E_C\hat{n}^2 + \frac{1}{2}E_J\hat{\phi}^2 - E_J(\cos\hat{\phi} + \frac{1}{2}\hat{\phi}^2) \quad (2.9)$$

where the first two terms corresponding to an LC circuit of capacitance C_Σ and inductance $E_J^{-1}(\Phi_0/2\pi)^2$, the linear part of the Josephson inductance. We have dropped the offset charge n_g on the basis that the frequency of the relevant low-lying energy levels are insensitive to this parameter. Importantly, although these energies are not sensitive to variations in n_g , it is still possible to use an external oscillating voltage source to cause transition between the transmon states. The last term in the Hamiltonian is the nonlinear correction to this harmonic potential which, for $E_J/E_C \gg 1$ and therefore $\Delta\hat{\phi} \ll 1$ can be truncated to its

first nonlinear correction

$$\hat{H}_q = 4E_C \hat{n}^2 + \frac{1}{2} E_J \hat{\phi}^2 - \frac{1}{4!} E_J \hat{\phi}^4. \quad (2.10)$$

As expected from the above discussion, the transmon is thus a weakly anharmonic oscillator. Note that in this approximation, the phase $\hat{\phi}$ is treated as a continuous variable with eigenvalues extending to arbitrary real values, rather than enforcing 2π -periodicity. This is allowed as long as the device is in a regime where $\hat{\phi}$ is sufficiently localized, which holds for low-lying energy eigenstates in the transmon regime with $E_J/E_C \gg 1$.

It is also useful to introduce creation and annihilation operators chosen to diagonalize the first two terms of \hat{H}_q . Denoting these operators \hat{b}^\dagger and \hat{b} , we have

$$\hat{\phi} = \left(\frac{2E_C}{E_J}\right)^{1/4} (\hat{b}^\dagger + \hat{b}) \quad (2.11)$$

$$\hat{n} = \frac{i}{2} \left(\frac{E_J}{2E_C}\right)^{1/4} (\hat{b}^\dagger - \hat{b}). \quad (2.12)$$

This form makes it clear that fluctuations of the phase $\hat{\phi}$ decrease with E_J/E_C , while the reverse is true for the conjugate charge \hat{n} . Using the expressions we just defined

$$\hat{H}_q = \sqrt{8E_C E_J} \hat{b}^\dagger \hat{b} - \frac{E_C}{12} (\hat{b}^\dagger + \hat{b})^4 \approx \hbar \omega_q \hat{b}^\dagger \hat{b} - \frac{E_C}{2} \hat{b}^\dagger \hat{b}^\dagger \hat{b} \hat{b} \quad (2.13)$$

where $\hbar \omega_q = \sqrt{8E_C E_J} - E_C$. In the second line, we have kept only the terms that have the same number of creation and annihilation operators. This is reasonable because, in a frame rotating at w_q , any terms with an unequal number of \hat{b} and \hat{b}^\dagger will be oscillating. If the frequency of these oscillations is larger than the prefactor of the oscillating term, then this term rapidly averages out and can be neglected. This rotating wave approximation is valid here if $\hbar \omega_q \gg E_C/4$ an inequality that is easily satisfied in the transmon regime.

The particular combination $\omega_p = \sqrt{8E_C E_J}/\hbar$ is known as the Josephson plasma frequency and corresponds to the frequency of small oscillations of the effective particle of mass C at the bottom of the well of the cosine potential of the Josephson junction. In the transmon regime, this frequency is renormalized by a 'Lamb shift' equal to the charging energy E_C such that $\omega_q = \omega_p - E_C/\hbar$ is the transition frequency between ground and the first excited state. Finally the last term of \hat{H}_q is a Kerr nonlinearity, with E_C/\hbar playing the role of Kerr frequency shift per excitation of the nonlinear oscillator. To see this even more clearly, it can be useful to rewrite \hat{H}_q as $\hat{H}_q = \hbar \tilde{\omega}_q (\hat{b}^\dagger \hat{b}) \hat{b}^\dagger \hat{b}$, where the frequency $\tilde{\omega}_q(\hat{b}^\dagger \hat{b}) = \omega_q - E_C(\hat{b}^\dagger \hat{b} - 1)/2\hbar$ of the oscillator is a decreasing function of the excitation number $\hat{b}^\dagger \hat{b}$. Considering only the first few levels of the transmon, this simply means that the transition frequency between the second and third excited states is smaller by E_C than

the transition frequency between the first and second excited states. In other words, in the regime of validity, the anharmonicity of the transmon is $-E_C$ with a typical value of $E_C/h \approx 100 - 400$ MHz.

While the nonlinearity E_C/h is small with respect to the oscillator frequency ω_q , it is in practice much larger than the spectral linewidth that can routinely be obtained for these artificial atoms and can therefore easily be spectrally resolved. As a result, and in contrast to more traditional realizations of Kerr nonlinearities in quantum optics, it is possible with superconducting quantum circuits to have a large Kerr nonlinearity even at the single-photon level. For quantum information processing, the presence of this nonlinearity is necessary to address only the ground and first excited state without unwanted transition to other states. In this case, the transmon acts as a two-level system, or a qubit. However, it is important to keep in mind that the transmon is a multilevel system and that is often necessary to include higher levels in the description of the device to quantitatively explain experimental observations. These higher levels can also be used to considerable advantage in some cases.

2.1.4 Flux-Tunable Transmons

A useful variant of the transmon artificial atom is the flux-tunable transmon, where the single Josephson junction is replaced with two parallel junctions forming a superconducting quantum interference device (SQUID). The transmon Hamiltonian then reads

$$\hat{H}_T = 4E_C\hat{n}^2 - E_{J1}\cos\hat{\phi}_1 - E_{J2}\cos\hat{\phi}_2 \quad (2.14)$$

where E_{Ji} is the Josephson energy of junction i , and $\hat{\phi}_i$ the phase difference across that junction. In the presence of an external flux Φ_x threading the SQUID loop and in the limit of negligible geometric inductance of the loop, flux quantization requires that $\hat{\phi}_1 - \hat{\phi}_2 = 2\pi\Phi_x/\Phi_0 \pmod{2\pi i}$. Defining the average phase difference $\hat{\phi} = (\hat{\phi}_1 + \hat{\phi}_2)/2$, the Hamiltonian can then be rewritten as

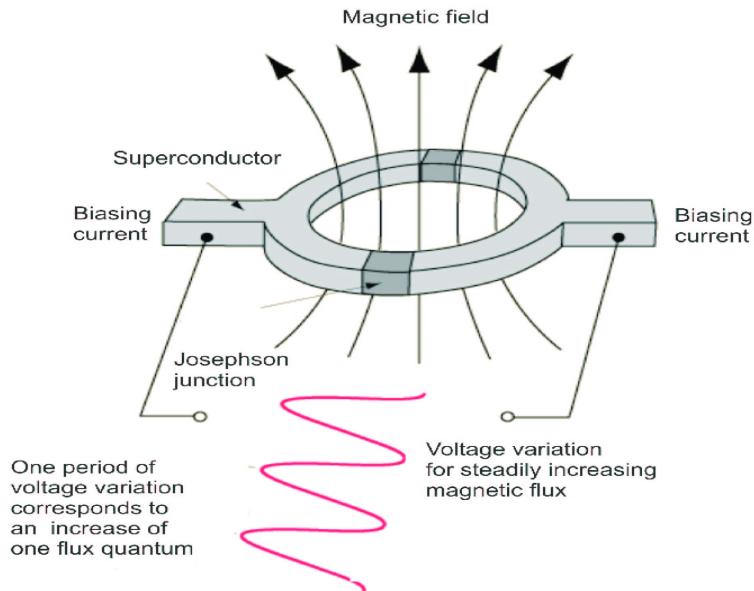
$$\hat{H}_T = 4E_C\hat{n}^2 - E_J(\Phi_x)\cos(\hat{\phi} - \phi_0), \quad (2.15)$$

where

$$E_J(\Phi_x) = E_{J\Sigma}\cos\left(\frac{\pi\Phi_x}{\Phi_0}\right)\sqrt{1 + d^2\tan^2\left(\frac{\pi\Phi_x}{\Phi_0}\right)} \quad (2.16)$$

with $E_{J\Sigma} = E_{J2} + E_{J1}$ and $d = (E_{J2} - E_{J1})/E_{J\Sigma}$ the junction asymmetry. The phase $\phi_0 = dtan(\pi\Phi_x/\Phi_0)$ can be ignored for a time-independent flux. Therefore, replacing the single junction with a SQUID loop yields an effective flux-tunable Josephson energy $E_J(\Phi_x)$ and in turn, this results in a flux-tunable transmon frequency $\omega_x(\Phi_x) = \sqrt{8E_C|E_J(\Phi_x)|} - E_C/\hbar$.

In practice, the transmon frequency can be tuned by as much as one GHz and as little as 10 – 20 ns. Dynamic range can also be traded for faster flux excursions by increasing the bandwidth of the flux bias lines.



As will become clear later, this additional control knob can lead to dephasing due to noise in the flux threading the SQUID loop. With this in mind, it is worth noticing that a larger asymmetry leads to a smaller range of tunability and thus also to less susceptibility to flux noise.

2.2 Light-Matter Interaction In Circuit QED

2.2.1 Exchange Interaction Between A Transmon And An Oscillator

Having introduced the two main characters of this review, the quantum harmonic oscillator and the transmon artificial atom, we are now ready to consider their interaction. Because of their large size coming from the requirement of having a low charging energy (large capacitance), transmon qubits can very naturally be capacitively coupled to microwave resonators. With resonators taking the place of the classical voltage source V_g , capacitive coupling to a resonator can be introduced in the transmon Hamiltonian with a quantized gate voltage $n_g \rightarrow -\hat{n}_r$, representing the charge bias of the transmon due to the resonator. The Hamiltonian of the combined system is therefore

$$\hat{H} = 4E_C(\hat{n} + \hat{n}_r)^2 - E_J \cos \hat{\phi} - \sum_m \hbar \omega_m \hat{a}_m^\dagger \hat{a}_m, \quad (2.17)$$

where $\hat{n}_r = \sum_m \hat{n}_m$ with $\hat{n}_m = (C_g/C_m)\hat{Q}_m/2e$ the contribution to the charge bias due to the m th resonator mode. Here, C_g is the gate capacitance and C_m the associated resonator mode capacitance. To simplify these expressions, we have assumed that $C_g \ll C_\Sigma, C_m$.

Assuming that the transmon frequency is much closer to one of the resonator modes than all the other modes, say $|\omega_0 - \omega_q| \ll |\omega_m - \omega_q|$ for $m \geq 1$, we truncate the sum over m to a single term. In this single-mode approximation, the Hamiltonian reduces to a single oscillator of frequency denoted ω_r coupled to a transmon. It is interesting to note that, regardless of the physical nature of the oscillator - for example a single mode of a 2D or 3D resonator - it is possible to represent this Hamiltonian with an equivalent circuit where the transmon is capacitively coupled to an LC oscillator. This type of formal representation of complex geometries in terms of equivalent lumped element circuits is generally known as "black-box quantization".

Using the creation and annihilation operators introduced before, in the single-mode approximation the Hamiltonian reduces to

$$\hat{H} \approx \hbar \omega_r \hat{a}^\dagger \hat{a} + \hbar \omega_q \hat{b}^\dagger \hat{b} - \frac{E_C}{2} \hat{b}^\dagger \hat{b}^\dagger \hat{b} \hat{b} - \hbar g (\hat{b}^\dagger - \hat{b}) (\hat{a}^\dagger - \hat{a}), \quad (2.18)$$

where ω_r is the frequency of the mode of interest and

$$g = \omega_r \frac{C_g}{C_\Sigma} \left(\frac{E_J}{2E_C} \right)^{1/4} \sqrt{\frac{\pi Z_r}{R_K}} \quad (2.19)$$

the oscillator-transmon, or light-matter, coupling constant. Here, Z_r is the characteristic impedance of the resonator mode and $R_K = h/e^2 \approx 25.8 \Omega$ the resistance quantum. The

above Hamiltonian can be simplified further in the experimentally relevant situation where the coupling constant is much smaller than the system frequencies, $|g| \ll \omega_r, \omega_q$. Invoking the rotating wave approximation, it simplifies to

$$\hat{H} \approx \hbar\omega_r \hat{a}^\dagger \hat{a} + \hbar\omega_q \hat{b}^\dagger \hat{b} - \frac{E_C}{2} \hat{b}^\dagger \hat{b}^\dagger \hat{b} \hat{b} + \hbar g (\hat{b}^\dagger \hat{a} + \hat{b} \hat{a}^\dagger) \quad (2.20)$$

The prefactor $(E_J/2E_C)^{1/4}$ is linked to the size of charge fluctuations in the transmon. By introducing a length l corresponding to the distance a Cooper pair travels when tunneling across the transmon's junction, it is tempting to interpret g as $\hbar g = d_0 \varepsilon_0$ with $d_0 = 2el(E_J/32E_C)^{1/4}$ the dipole moment of the transmon and $\varepsilon_0 = (\omega_r/l)(C_g/C_\Sigma)\sqrt{\hbar Z_r}/2$ the resonator's zero-point electric field as seen by the transmon. Since the two factors can be made large, especially so in the transmon regime where $d_0 \gg 2el$, the electric-dipole interaction strength g can be made very large, much more so than with natural atoms in cavity QED. It is also instructive to express g as

$$g = \omega_r \frac{C_g}{C_\Sigma} \left(\frac{E_J}{2E_C} \right)^{1/4} \sqrt{\frac{Z_r}{Z_{vac}}} \sqrt{2\pi\alpha} \quad (2.21)$$

where $\alpha = Z_{vac}/2R_K$ is the fine-structure constant and $Z_{vac} = \sqrt{\mu_0/\epsilon_0} \approx 377\Omega$ the impedance of vacuum with ϵ_0 the vacuum permittivity and μ_0 the vacuum permeability. To find α here should not be surprising because this quantity characterizes the interaction between the electromagnetic field and charged particles. Here, this interaction is reduced by the fact that both Z_r/Z_{vac} and C_g/C_Σ are smaller than unity. Very large couplings can nevertheless be achieved by working with large values of E_J/E_C or, in other words, in the transmon regime. Large g is therefore obtained at the expense of reducing the transmon's relative anharmonicity. We note that the coupling can be increased by boosting the resonator's impedance, something that can be realized, for example, by replacing the resonator's center conductor with a junction array.

Apart from a change in the details of the expression of coupling g , the above discussion holds for transmons coupled to lumped, 2D or 3D resonators. Importantly, by going from 2D to 3D, the resonator mode volume is made significantly larger leading to an important reduction in the vacuum fluctuations of the electric field. This can be made without change in the magnitude of g simply by making the transmon larger thereby increasing its dipole moment. The transmon then essentially becomes an antenna that is optimally placed within the 3D resonator to strongly couple to one of the resonator modes.

To strengthen the analogy with cavity QED even further, it is useful to restrict the description of the transmon to its first two levels. This corresponds to making the replacements $\hat{b}^\dagger \rightarrow \hat{\sigma}_+ = |e\rangle\langle g|$ and $\hat{b} \rightarrow \hat{\sigma}_- = |g\rangle\langle e|$ to obtain the well-known Jaynes-Cummings Hamiltonian

$$\hat{H}_{JC} = \hbar\omega_r \hat{a}^\dagger \hat{a} + \frac{\hbar\omega_q}{2} \hat{\sigma}_z + \hbar g (\hat{a}^\dagger \hat{\sigma}_- + \hat{a} \hat{\sigma}_+), \quad (2.22)$$

where we use the convention $\hat{\sigma}_z = |e\rangle\langle e| - |g\rangle\langle g|$. The last term of this Hamiltonian describes the coherent exchange of a single quantum between light and matter, here realized as a photon in the oscillator or an excitation of the transmon.

2.2.2 The Jaynes-Cummings Spectrum

The Jaynes-Cummings Hamiltonian is an exactly solvable model which very accurately describes many situations in which an atom, artificial or natural, can be considered as a two-level system in interaction with a single mode of the electromagnetic field. This model can yield qualitative agreement with experiments in situations where only the first two levels of the transmon, $|\sigma = \{g, e\}\rangle$, play an important role. It is often the case, however, that quantitative agreement between theoretical predictions and experiments is obtained only when accounting for higher transmon energy levels and the multimode nature of the field. Nevertheless, since a great deal of insight can be gained from this.

In the absence of coupling g , the bare states of the qubit-field system are labelled σ, n with σ defined above and n the photon number. The dressed eigenstates of the Jaynes-Cummings Hamiltonian, $|\overline{\sigma, n}\rangle = \hat{U}^\dagger |\sigma, n\rangle$, can be obtained from these bare states using Bogoliubov-like unitary transformation

$$\hat{U} = \exp[\Lambda(\hat{N}_T)(\hat{a}^\dagger \hat{\sigma}_- - \hat{a} \hat{\sigma}_+)], \quad (2.23)$$

where we have defined

$$\Lambda(\hat{N}_T) = \frac{\arctan(2\lambda\sqrt{\hat{N}_T})}{2\sqrt{\hat{N}_T}}. \quad (2.24)$$

Here, $\hat{N}_T = \hat{a}^\dagger \hat{a} + \hat{\sigma}_+ \hat{\sigma}_-$ is the operator associated with the total number of excitations, which commutes with \hat{H}_{JC} , and $\lambda = g/\Delta$ with $\Delta = \omega_q - \omega_r$ the qubit-resonator detuning. Under this transformation, \hat{H}_{JC} takes the diagonal form

$$\hat{H}_D = \hat{U}^\dagger \hat{H}_{JC} \hat{U} = \hbar\omega_r \hat{a}^\dagger \hat{a} + \frac{\hbar\omega_q}{2} \hat{\sigma}_z - \frac{\hbar\Delta}{2} (1 - \sqrt{1 + 4\lambda^2 \hat{N}_T}) \hat{\sigma}_z. \quad (2.25)$$

The dressed-state energies can be read directly from this expression and the Jaynes-Cummings spectrum consists of doublets $\{|\overline{g, n}\rangle, |\overline{e, n-1}\rangle\}$ of fixed excitation number

$$\begin{aligned} E_{\overline{g,n}} &= \hbar n \omega_r - \frac{\hbar}{2} \sqrt{\Delta^2 + 4g^2 n}, \\ E_{\overline{e,n-1}} &= \hbar n \omega_r + \sqrt{\Delta^2 + 4g^2 n}, \end{aligned} \quad (2.26)$$

and of ground state $|\overline{g, 0}\rangle = |g, 0\rangle$ of energy $E_{\overline{g,0}} = -\hbar\omega_q/2$. The excited dressed states are

$$|\overline{g, n}\rangle = \cos(\theta_n/2) |g, n\rangle - \sin(\theta_n/2) |e, n-1\rangle, \quad (2.27)$$

$$|\overline{g, n-1}\rangle = \sin(\theta_n/2) |g, n\rangle + \cos(\theta_n/2) |e, n-1\rangle, \quad (2.28)$$

with $\theta_n = \arctan(2g\sqrt{n}/\Delta)$.

A crucial feature of this energy spectrum is the scaling with the photon number n . In particular, for zero detuning, $\Delta = 0$, the energy levels $E_{g,n}$ and $E_{e,n-1}$ are split by $2g\sqrt{n}$, in contrast to two coupled harmonic oscillators where the energy splitting is independent of n . Experimentally probing this spectrum thus constitutes a way to assess the quantum nature of the coupled system.

2.2.3 Dispersive Regime

On resonance, $\Delta = 0$, the dressed-states are maximally entangled qubit-resonator states implying that the qubit is, by itself, never in a well-defined state, meaning that the reduced state of the qubit found by tracing over the resonator is not pure. For quantum information processing, it is therefore more practical to work in the dispersive regime where the qubit-resonator detuning is large with respect to the coupling strength, $|\lambda| = |g/\Delta| \ll 1$. In this case, the coherent exchange of a quanta between the two systems described by the last term of \hat{H}_{JC} is not resonant, and interactions take place only via photon processes. Qubit and resonator are therefore only weakly entangled and a simplified model obtained below from second-order perturbation theory is often an excellent approximation.

2.2.3.1 Schrieffer-Wolff Approach and Bogoliubov Approach

To find an approximation to Eq. 2.20 valid in the dispersive regime, we perform a Schrieffer-Wolff transformation to second order. As long as the interaction term in Eq. 2.20 is sufficiently small, the resulting effective Hamiltonian is well approximated by

$$\begin{aligned} \hat{H}_{\text{disp}} \simeq & \hbar\omega_r \hat{a}^\dagger \hat{a} + \hbar\omega_q \hat{b}^\dagger \hat{b} - \frac{E_C}{2} \hat{b}^\dagger \hat{b}^\dagger \hat{b} \hat{b} \\ & + \sum_{j=0}^{\infty} h (\Lambda_j + \chi_j \hat{a}^\dagger \hat{a}) |j\rangle \langle j| \end{aligned} \quad (2.29)$$

where $|j\rangle$ label the eigenstates of the transmon which, under the approximation used to obtain Eq. 2.13, are just the eigenstates of the number operator $\hat{b}^\dagger \hat{b}$. Moreover, we have defined

$$\begin{aligned} \Lambda_j &= \chi_{j-1,j} \quad \chi_j = \chi_{j-1,j} - \chi_{j,j+1}, \\ \chi_{j-1,j} &= \frac{jg^2}{\Delta - (j-1)E_C/\hbar} \end{aligned} \quad (2.30)$$

for $j > 0$ and $\Lambda_0 = 0$, $\chi_0 = -g^2/\Delta$. Here χ_j are known as dispersive shifts, while Λ_j are Lamb shifts and are signatures of vacuum fluctuations. Truncating Eq. 2.29 to the first two levels of the transmon leads to the more standard form of the dispersive Hamiltonian.

$$\hat{H}_{\text{disp}} \approx \hbar\omega'_r \hat{a}^\dagger \hat{a} + \frac{\hbar\omega'_q}{2} \hat{\sigma}_z + \hbar\chi \hat{a}^\dagger \hat{a} \hat{\sigma}_z, \quad (2.31)$$

where χ is the qubit state-dependent dispersive cavity shift with

$$\begin{aligned} \omega'_r &= \omega_r - \frac{g^2}{\Delta - E_C/\hbar}, & \omega'_q &= \omega_q + \frac{g^2}{\Delta}, \\ \chi &= -\frac{g^2 E_C/\hbar}{\Delta(\Delta - E_C/\hbar)}. \end{aligned} \quad (2.32)$$

These dressed frequencies are what are measured experimentally in the dispersive regime. It is important to emphasize that the frequencies entering the right-hand-sides of Eq. 2.32 are the *bare* qubit and resonator frequencies. We note that the Schrieffer-Wolff transformation also gives rise to resonator and qubit self-Kerr nonlinearities at fourth order. As already mentioned, these perturbative results are valid when the interaction term in 2.20 is sufficiently small compared to the energy splitting of the bare transmon-oscillator energy levels, $|\lambda| = |g/\Delta| \ll 1$. Because the matrix elements of the operators involved in the interaction term scale with the number of photons in the resonator and the number of qubit excitations, a more precise bound on the validity of Eq. 2.29 needs to take into account these quantities. We find that for the above second order perturbation results to be good approximation, the oscillator photon number \bar{n} should be much smaller than a critical photon number n_{crit}

$$\bar{n} \ll n_{\text{crit}} \equiv \frac{1}{2j+1} \left(\frac{|\Delta - jE_C/\hbar|^2}{4g^2} - j \right) \quad (2.33)$$

where $j = 0, 1, \dots$ refers to the qubit state as before. For $j = 0$, this yields the familiar value $n_{\text{crit}} = (\Delta/2g)^2$ for the critical photon number expected from the Jaynes-Cummings model, while setting $j = 1$ gives a more conservative bound. In either case, this gives only a rough estimate for when to expect higher-order effects to become important. It is worth contrasting Eq. 2.32 to the results expected from performing a dispersive approximation to the Jaynes-Cummings model Eq. 2.22, which leads to $\chi = g^2/\Delta$. This agrees above result in the limit of very large E_C but, since E_C/\hbar is typically rather small compared to Δ in most transmon experiments, the two-level system Jaynes-Cummings model gives a poor prediction for the dispersive shift χ in practice. The intuition here is that E_C determines the anharmonicity of the transmon. Two coupled harmonic oscillators can shift each other's frequencies, but only in a state-independent manner. Thus the dispersive shift must vanish in the limit of E_C going to zero.

We now present an approach to arrive at Eq. 2.31 that can be simpler than performing a Schrieffer-Wolff transformation and which is often used in the circuit QED literature. To proceed, it is convenient to write Eq. 2.20 as the sum of a linear and nonlinear part, $\hat{H} = \hat{H}_L + \hat{H}_{NL}$, where

$$\hat{H}_L = \hbar\omega_r \hat{a}^\dagger \hat{a} + \hbar\omega_q \hat{b}^\dagger \hat{b} + \hbar g \left(\hat{b}^\dagger \hat{a} + \hat{b} \hat{a}^\dagger \right) \quad (2.34)$$

$$\hat{H}_{NL} = -\frac{E_C}{2}\hat{b}^\dagger\hat{b}^\dagger\hat{b}\hat{b} \quad (2.35)$$

The linear part \hat{H}_L can be diagonalized exactly using the Bogoliubov transformation

$$\hat{U}_{disp} = \exp \left[\Lambda \left(\hat{a}^\dagger \hat{b} - \hat{a} \hat{b}^\dagger \right) \right] \quad (2.36)$$

Under this unitary, the annihilation operators transform as $\hat{U}_{disp}^\dagger \hat{a} \hat{U}_{disp} = \cos(\Lambda)\hat{a} + \sin(\Lambda)\hat{b}$ and $\hat{U}_{disp}^\dagger \hat{b} \hat{U}_{disp} = \cos(\Lambda)\hat{b} - \sin(\Lambda)\hat{a}$. With the choice $\Lambda = \frac{1}{2} \arctan(2\lambda)$, this results in the diagonal form

$$\hat{U}_{disp}^\dagger \hat{H}_L \hat{U}_{disp} = \hbar\tilde{\omega}_r \hat{a}^\dagger \hat{a} + \hbar\omega_r \hat{b}^\dagger \hat{b}, \quad (2.37)$$

with the dressed frequencies

$$\bar{\omega}_r = \frac{1}{2} \left(\omega_r + \omega_r - \sqrt{\Delta^2 + 4g^2} \right), \quad \tilde{\omega}_r = \frac{1}{2} \left(\omega_r + \omega_n + \sqrt{\Delta^2 + 4g^2} \right). \quad (2.38)$$

Applying the same transformation to \hat{H}_{NL} and, in the dispersive regime, expanding the result in orders of λ leads to the dispersive Hamiltonian.

$$\begin{aligned} \hat{H}_{disp} &= \hat{U}_{disp}^\dagger \hat{H} \hat{U}_{disp} \\ &\simeq \hbar\tilde{\omega}_r \hat{a}^\dagger \hat{a} + \hbar\tilde{\omega}_q \hat{b}^\dagger \hat{b} \\ &+ \frac{\hbar K_a}{2} \hat{a}^\dagger \hat{a}^\dagger \hat{a} \hat{a} + \frac{\hbar K_b}{2} \hat{b}^\dagger \hat{b}^\dagger \hat{b} \hat{b} + h\chi_{ab} \hat{a}^\dagger \hat{a} \hat{b}^\dagger \hat{b}, \end{aligned} \quad (2.39)$$

where we have introduced

$$\begin{aligned} K_a &\simeq -\frac{E_C}{2h} \left(\frac{g}{\Delta} \right)^4, \quad K_b \simeq -E_C/\hbar, \\ \chi_{ab} &\simeq -2 \frac{g^2 E_C / \hbar}{\Delta (\Delta - E_C / \hbar)} \end{aligned} \quad (2.40)$$

The first two of these quantities are self-Kerr nonlinearities, while the third is a cross-Kerr interaction. All are negative in the dispersive regime. The above expression for χ_{ab} is obtained after performing a Schrieffer-Wolff transformation to eliminate a term of the form $\hat{b}^\dagger \hat{b} \hat{a}^\dagger \hat{b} + H.c.$ that results from applying U_{disp} on H_{NL} . Higher-order terms in λ and other terms rotating at frequency Δ or faster have been dropped to arrive at Eq.2.39. Truncating Eq.2.39 to the first two levels of the transmon correctly leads to Eqs. 2.31 and 2.32. Importantly, these expressions are not valid if the excitation number of the resonator or the transmon is too large or if $|\Delta| \sim E_C/\hbar$. The regime $0 < \Delta < E_C$, known as the straddling regime, is qualitatively different from the usual dispersive regime. It is characterized by positive self-Kerr and cross-Kerr nonlinearities, $K_a, \chi_{ab} > 0$, and is better addressed by exact numerical diagonalization of Eq.2.17. A remarkable feature of circuit QED is the large nonlinearities that are achievable in the dispersive regime. Dispersive shifts larger than the

resonator or qubit linewidth, $\chi > \kappa, \gamma$, are readily realized in experiments, a regime referred to as strong dispersive coupling. It is also possible to achieve large self-Kerr nonlinearities for the resonator, $K_a > \kappa$. These nonlinearities can be enhanced by embedding Josephson junctions in the center conductor of the resonator, an approach which is used for example in quantum-limited parametric amplifiers or for the preparation of quantum states of the microwave electromagnetic field.

2.2.4 Josephson Junctions Embedded In Multimode Electromagnetic Environments

So far, we have focused on the capacitive coupling of a transmon to a single mode of an oscillator. For many situations of experimental relevance it is, however, necessary to consider the transmon, or even multiple transmons, embedded in an electromagnetic environment with a possibly complex geometry, such as an 3D cavity.

Consider the situation where a capacitively shunted Josephson junction is embedded in some electromagnetic environment represented by the impedance $Z(\omega)$. To keep the discussion simple, we consider here a single junction but the procedure can easily be extended to multiple junctions. As discussed earlier, the Hamiltonian of the shunted junction can be decomposed into a linear term of capacitance $C_\Sigma = C_S + C_J$ and linear inductance $L_J = E_J^{-1}(\Phi_0/2\pi)^2$, and a purely nonlinear element.

We assume that the electromagnetic environment is linear, nonmagnetic and has no free charges and current. Since C_Σ and L_J are themselves linear elements, we might as well consider them part of the electromagnetic environment too. Combining all linear contributions, we write a Hamiltonian for the entire system, junction plus the surrounding electromagnetic environment, as $\hat{H} = \hat{H}_L + \hat{H}_{NL}$ with

$$\hat{H}_{NL} = -E_J(\cos\hat{\phi} + \frac{1}{2}\hat{\phi}^2) \quad (2.41)$$

the nonlinear part of the transmon Hamiltonian already introduced before. A good strategy is to first diagonalize the linear part, \hat{H}_L . Subsequently, the phase difference $\hat{\phi}$ across the junction can be expressed as a linear combination of the eigenmodes of \hat{H}_L , a decomposition which is then used in \hat{H}_{NL} .

A convenient choice of canonical fields for the electromagnetic environment are the electric displacement field $\hat{D}(x)$ and the magnetic field $\hat{B}(x)$, which can be expressed in terms of bosonic creation and annihilation operators

$$\hat{D}(x) = \sum_m [D_m(x)\hat{a}_m + H.c], \quad (2.42)$$

$$\hat{B}(x) = \sum_m [B_m(x)\hat{a}_m + H.c], \quad (2.43)$$

where $[\hat{a}_m, \hat{a}_m^\dagger] = \delta_{mn}$. The more commonly used electric field is related to the displacement field through $\hat{D}(x) = \epsilon_0 \hat{E}(x) + \hat{P}(x)$, where $\hat{P}(x)$ is the polarization of the medium. Moreover, the mode functions $D_m(x)$ and $B_m(x)$ can be chosen to satisfy the orthogonality and normalization conditions such that

$$\hat{H}_L = \sum_m \hbar\omega_m \hat{a}_m^\dagger \hat{a}_m. \quad (2.44)$$

We have implicitly assumed that the eigenmodes form a discrete set. If some part of the spectrum is continuous, which is the case for infinite systems such as open waveguides, the sums must be replaced by integrals over the relevant frequency ranges. The result is very general, holds for arbitrary geometries, and mirrors, and materials with dispersion.

Diagonalizing \hat{H}_L amounts to determining the mode functions $\{\hat{D}_m(x), \hat{B}_m(x)\}$, which is essentially a classical electromagnetism problem that can be approached using numerical software such as finite element solvers. Assuming that the mode functions have been found, we now turn to \hat{H}_{NL} for which we relate $\hat{\phi}$ to the bosonic operators \hat{a}_m . This can be done by noting again that $\hat{\phi}(t) = 2\pi \int dt' \hat{V}(t')/\Phi_0$, where the voltage is simply the line integral of the electric field $\hat{V}(t) = \int dl \cdot \hat{D}(x)/\epsilon$ across the junction. Consequently, the phase variable can be expressed as

$$\hat{\phi} = \sum_m [\phi_m \hat{a}_m + H.c], \quad (2.45)$$

where $\phi_m = i(2\pi/\Phi_0) \int_{x'_J}^{x_J} dl \cdot \hat{D}_m(x)/(\omega_m \epsilon)$ is the dimensionless magnitude of the zero-point fluctuations of the m th mode as seen by the junction and the boundary conditions.

Using the phase variable definition in \hat{H}_{NL} we expand the cosine to fourth order. This means that the capacitively shunted junction is well in the transmon regime, with a small anharmonicity relative to Josephson energy. Focusing on the dispersive regime where all eigenfrequencies ω_m are sufficiently well separated, and neglecting fast-rotating terms leads to

$$\hat{H}_{NL} \approx \sum_m \hbar \Delta_m \hat{a}_m^\dagger \hat{a}_m + \frac{1}{2} \sum_m \hbar K_m (\hat{a}_m^\dagger)^2 \hat{a}_m^2 + \sum_{m>n} \hbar \chi_{m,n} \hat{a}_m^\dagger \hat{a}_m \hat{a}_n^\dagger \hat{a}_n \quad (2.46)$$

where $\Delta_m = \frac{1}{2} \sum_n \chi_{m,n}$, $K_m = \frac{\chi_{m,n}}{2}$ and $\hbar \chi_{m,n} = -E_J \phi_m^2 \phi_n^2$. It is also useful to introduce the energy participation ratio p_m , defined to be the fraction of the total inductive energy of mode m that is stored in the junction $p_m = (2E_J/\hbar\omega_m)\phi_m^2$ such that we can write

$$\chi_{m,n} = -\frac{\hbar\omega_m\omega_n}{4E_J} p_m p_n. \quad (2.47)$$

As is clear from the above discussion, finding the nonlinear Hamiltonian can be replaced to finding the eigenmodes of the system and the zero-point fluctuations ϕ_m , can be complicated for a complex geometry. As already mentioned this is, however, an entirely classical electromagnetism problem.

An alternative approach is to represent the linear electromagnetic environment seen by the purely nonlinear element as an impedance $Z(\omega)$. Neglecting loss, any such impedance can be represented by an equivalent circuit of possibly infinitely many LC oscillators connected in series. The eigenfrequencies $\hbar\omega_m = 1/\sqrt{L_m C_m}$, can be determined by the real parts of the zeros of the admittance $Y(\omega) = Z^{-1}(\omega)$, and the effective impedance of the m th mode as seen by the junction can be found from $Z_m^{eff} = 2/[\omega_m Im Y'(\omega_m)]$. The effective impedance is related to the zero-point fluctuations used above as $Z_m^{eff} = 2(\Phi_0/2\pi)^2 \phi_m^2 / \hbar = R_K \phi_m^2 / (4\pi)$. From this point of view, the quantization procedure thus reduced to the task of determining the impedance $Z(\omega)$ as a function of frequency.

2.2.4.1 Beyond the transmon: multilevel artificial atom

In the preceding sections, we have relied on a perturbative expansion of the cosine potential of the transmon under the assumption $E_J/E_C \gg 1$. To go beyond this regime one can instead resort to exact diagonalization of the transmon Hamiltonian. Returning to the full transmon-resonator Hamiltonian Eq. 2.17, we write

$$\begin{aligned} \hat{H} &= 4E_C \hat{n} - E_J \cos \hat{\phi} + \hbar\omega_r \hat{a}^\dagger \hat{a} + 8E_C \hat{n} \hat{n}_r \\ &= \sum_j \hbar\omega_j |j\rangle \langle j| + \hbar\omega_r \hat{a}^\dagger \hat{a} + i \sum_{ij} \hbar g_{ij} t |i\rangle \langle j| (\hat{a}^\dagger - \hat{a}) \end{aligned} \quad (2.48)$$

where $|j\rangle$ are now the eigenstates of the bare transmon Hamiltonian $\hat{H}_T = 4E_C \hat{n} - E_J \cos \hat{\phi}$ obtained from numerical diagonalization and we have defined

$$\hbar g_{ij} = 2e \frac{C_g}{CC_\Sigma} Q_{zpf} \langle i | \hat{n} | j \rangle \quad (2.49)$$

The eigenfrequencies ω_j and the matrix elements $\langle i | \hat{n} | j \rangle$ can be computed numerically in the charge basis. Alternatively, they can be determined by taking advantage of the fact that, in the phase basis, Eq. 2.8 takes the form of a Mathieu equation whose exact solution is known. The second form of Eq. 2.48 written in terms of energy eigenstates $|j\rangle$ is a very general Hamiltonian that can describe an arbitrary multilevel artificial atom capacitively coupled to a resonator. Similarly to the discussion of dispersive regime section, in the dispersive regime where $|g_{ij}| \sqrt{n+1} \ll |\omega_i - \omega_j - \omega_r|$ for all relevant atomic transitions $i \rightarrow j$ and with n the oscillator photon number, it is possible to use a Schrieffer-Wolff transformation

to approximately diagonalize Eq. 2.48. To second order one finds

$$\begin{aligned}\hat{H} \simeq & \sum_j \hbar (\omega_j + \Lambda_j) |j\rangle\langle j| + \hbar\omega_r \hat{a}^\dagger \hat{a} \\ & + \sum_j \hbar x_j \hat{a}^\dagger \hat{a} |j\rangle\langle j|,\end{aligned}\quad (2.50)$$

where

$$\begin{aligned}\hat{H} \simeq & \sum_j \hbar (\omega_j + \Lambda_j) |j\rangle\langle j| + \hbar\omega_r \hat{a}^\dagger \hat{a} \\ & + \sum_j \hbar x_j \hat{a}^\dagger \hat{a} |j\rangle\langle j|,\end{aligned}\quad (2.51)$$

This result is, as already stated, very general, and can be used with a variety of artificial atoms coupled to a resonator in the dispersive limit.

2.2.4.2 Alternative coupling schemes

Coupling the electric dipole moment of a qubit to the zero-point electric field of an oscillator is the most common approach to light-matter coupling in a circuit but it is not the only possibility. Another approach is to take advantage of the mutual inductance between a flux qubit and the center conductor of resonator to couple the qubit's magnetic dipole to the resonator's magnetic field. Stronger interaction can be obtained by galvanically connecting the flux qubit to the center conductor of a transmission-line resonator. In such a situation, the coupling can be engineered to be as large, or even larger, than the system frequencies allowing to reach what is known at the ultrastrong coupling regime. Yet another approach is to couple the qubit to the oscillator in such a way that the resonator field does not result in qubit transitions but only shifts the qubit's frequency. This is known as longitudinal coupling and is represented by the Hamiltonian.

$$\hat{H}_x = \hbar\omega a^\dagger a + \frac{\hbar\omega_g}{2} \hat{\sigma} = +hg_z (a^\dagger + \hat{a}) \hat{\sigma}_z. \quad (2.52)$$

Because light-matter interaction in \hat{H}_z is proportional to $\hat{\sigma}_z$ rather than $\hat{\sigma}_x$, the longitudinal interaction does not lead to dressing of the form discussed in the Jaynes-Cummings spectrum section. Some of the consequences of this observation, in particular for qubit readout, are discussed in other approaches of dispersive qubit readout section.

2.2.5 Coupling To The Outside World: The Role Of The Environment

So far we have dealt with isolated quantum systems. A complete description of quantum electrical circuits, however, must also take into account a description of how these systems couple to their environment, including any measurement apparatus and control circuitry. In fact, the environment plays a dual role in quantum technology: Not only is a description of

quantum systems as perfectly isolated unrealistic, as coupling to unwanted environmental degrees of freedom is unavoidable, but a perfectly isolated system would also not be very useful since we would have no means of controlling or observing it. For these reasons, in this section we consider quantum systems coupled to external transmission lines.

2.2.5.1 Wiring Up Quantum Systems With Transmission Lines

We start the discussion by considering transmission lines coupled for losses and can be used to apply and receive quantum signals for control and measurement. To be specific, we consider a semi-infinite coplanar waveguide transmission line capacitively coupled at one end to an oscillator. The semi-infinite transmission line can be considered as a limit of the coplanar waveguide resonator of finite length where one of the boundaries is now pushed to infinity, $d \rightarrow \infty$. Increasing the length of the transmission line leads to a densely packed frequency spectrum, which in its infinite limit must be treated as a continuum. The Hamiltonian of the transmission line is consequently

$$\hat{H}_{t\text{ml}} = \int_0^\infty d\omega \hbar \omega \hat{b}_\omega^\dagger \hat{b}_\omega \quad (2.53)$$

where the mode operators now satisfy $[\hat{b}_\omega, \hat{b}_\omega^\dagger] = \delta(\omega - \omega')$. Similarly, the position-dependent flux and charge operators of the transmission line are given in the continuum limit by

$$\hat{\Phi}_{t\text{ml}}(x) = \int_0^\infty d\omega \sqrt{\frac{\hbar}{\pi \omega c v}} \cos\left(\frac{\omega x}{v}\right) (\hat{b}_\omega^\dagger + \hat{b}_\omega) \quad (2.54)$$

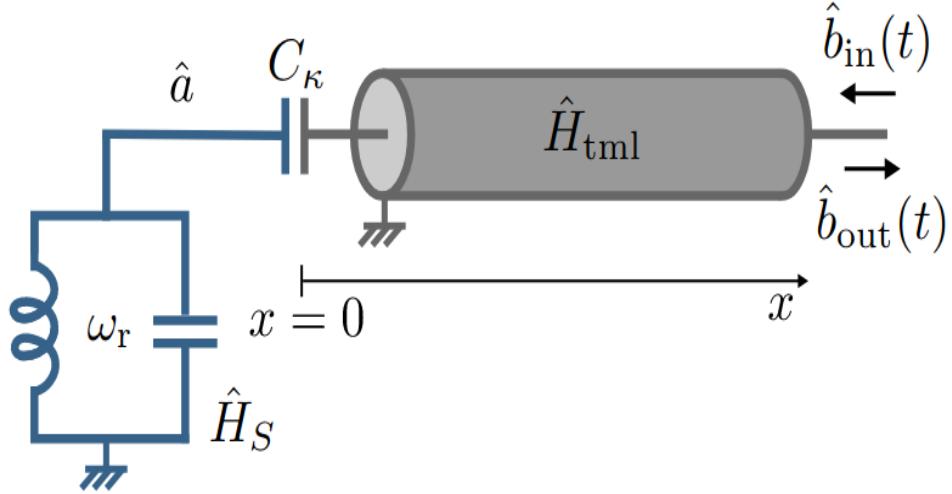
$$\hat{Q}_{t\text{ml}}(x) = i \int_0^\infty d\omega \sqrt{\frac{\hbar \omega c}{\pi v}} \cos\left(\frac{\omega x}{v}\right) (\hat{b}_\omega^\dagger - \hat{b}_\omega) \quad (2.55)$$

These are the canonical fields of the transmission line and in the Heisenberg picture, are related through $\hat{Q}_{t\text{ml}}(x, t) = c \hat{\Phi}_{t\text{ml}}(x, t)$. In these expressions, $v = 1/\sqrt{lc}$ is the speed of light in the transmission line, with c and l the capacitance and inductance per unit length, respectively.

Considering capacitive coupling of the line to the oscillator at $x = 0$, the total Hamiltonian takes the form

$$\hat{H} = \hat{H}_S + \hat{H}_{t\text{ml}} - \hbar \int_0^\infty d\omega \lambda(\omega) (\hat{b}_\omega^\dagger - \hat{b}_\omega) (\hat{a}^\dagger - \hat{a}), \quad (2.56)$$

where $\hat{H}_S = \hbar \omega_r \hat{a}^\dagger \hat{a}$ is the oscillator Hamiltonian. Moreover, $\lambda(\omega) = (C_K / \sqrt{c C_r}) \sqrt{\omega_r \omega / 2\pi v}$ is the frequency-dependent coupling strength, with C_K the coupling capacitance and C_r the resonator capacitance. These expressions neglect small renormalizations of the capacitances due to C_K .



In the following, $\lambda(\omega)$ is assumed sufficiently small compared to ω_r , such that the interaction can be treated as a perturbation. In this situation, the system's Q factor is large and the oscillator only responds in a small bandwidth around ω_r . It is therefore reasonable to take $\lambda(\omega) \approx \lambda(\omega_r)$. Dropping the rapidly oscillating terms finally leads to

$$\hat{H} \approx \hat{H}_S + \hat{H}_{tml} + \hbar \int_0^\infty d\omega \lambda(\omega_r) (\hat{a}\hat{b}_\omega^\dagger - \hat{a}^\dagger\hat{b}_\omega). \quad (2.57)$$

Under the well-established Born-Markov approximations, the above equation leads to Lindblad-form Markovian master equation for the system's density matrix ρ

$$\dot{\rho} = -i[\hat{H}_S, \rho] + \kappa(\bar{n}_\kappa + 1)\mathcal{D}[\hat{a}]\rho + \kappa\bar{n}_\kappa\mathcal{D}[\hat{a}^\dagger]\rho \quad (2.58)$$

where $\kappa = 2\pi\lambda(\omega_r)^2 = Z_{tml}\omega_r^2 C_\kappa^2 / C_r$ is the photon decay rate, or linewidth, of the oscillator introduced earlier. Moreover, $\bar{n}_\kappa = \bar{n}_\kappa(\omega_r)$ is the number of thermal photons of the transmission line as given by the Bose-Einstein distribution, $\langle \hat{b}_\omega^\dagger \hat{b}_{\omega'} \rangle = \bar{n}_\kappa(\omega)\delta(\omega - \omega')$, at the system frequency ω_r and environment temperature T . The symbol $\mathcal{D}[\hat{\mathcal{O}}] \bullet$ represents the dissipator

$$\mathcal{D}[\hat{\mathcal{O}}] \bullet = \hat{\mathcal{O}} \bullet \hat{\mathcal{O}}^\dagger - \frac{1}{2}\{\hat{\mathcal{O}}^\dagger \hat{\mathcal{O}}, \bullet\}, \quad (2.59)$$

with $\{.,.\}$ the anticommutator. Focussing on the second term, the role of this superoperator can be understood intuitively by noting that the term $\hat{\mathcal{O}}\rho\hat{\mathcal{O}}^\dagger$ with $\hat{\mathcal{O}} = \hat{a}$ acts on the Fock state $|n\rangle$ as $\hat{a}|n\rangle\langle n|\hat{a}^\dagger = n|n-1\rangle\langle n-1|$. The second term of $\dot{\rho}$ therefore corresponds to photon loss at rate κ . Finite temperature stimulates photon emission, boosting the loss rate to $\kappa(\bar{n}_\kappa + 1)$. On the other hand, the last term of $\dot{\rho}$ corresponds to absorption of thermal

photons by the system. Because $\hbar\omega_r \gg k_B T$ at dilution refrigerator temperatures, it is often assumed that $\bar{n}_\kappa \rightarrow 0$. Deviations from this expected behaviour are, however, common in practice due to residual thermal radiation propagating along control lines connecting to room temperature equipment and to uncontrolled sources of heating. Approaches to mitigate this problem using absorptive components are being developed.

2.2.5.2 Input-Output Theory In Electrical Networks

While the master equation describes the system's damped dynamics, it provides no information on the fields radiated by the system. Since radiated signals are what are measured experimentally, it is of practical importance to include those in our model.

This is known as the input-output theory for which two standard approaches exist. The first is to work directly with the Hamiltonian equation and consider Heisenberg picture equations of motion for the system and field annihilation operators \hat{a} and \hat{b}_ω .

An alternative approach is to introduce a decomposition of the transmission line modes in terms of left and right moving fields, linked by a boundary condition at the position of the oscillator which we take to be $x = 0$ with the transmission line at $x \geq 0$. The advantage of this approach is that the oscillator's input and output fields are then defined in terms of easily identifiable left and right moving radiation field components propagating along the transmission line. To achieve this, we replace the modes $\cos(\omega x/v)\hat{b}_\omega$ in $\hat{\Phi}_{tml}(x)$ and $\hat{Q}_{tml}(x)$ by $(\hat{b}_{R\omega}e^{i\omega x/v} + \hat{b}_{L\omega}e^{-i\omega x/v})/2$. Since the number of degrees of freedom of the transmission line has seemingly doubled, the modes $\hat{b}_{L/R\omega}$ cannot be independent. Indeed, the dynamics of one set of modes is fully determined by the other set through a boundary condition linking the left and right movers at $x = 0$.

To see this, it is useful to first decompose the voltage $\hat{V}(x, t) = \dot{\hat{\Phi}}_{tml}(x, t)$ at $x = 0$ into left moving (input) and right moving (output) contributions as $\hat{V}(t) = \hat{V}(x = 0, t) = \hat{V}_{in}(t) + \hat{V}_{out}(t)$, where

$$\hat{V}_{in/out}(t) = i \int_0^\infty d\omega \sqrt{\frac{\hbar\omega}{4\pi cv}} e^{i\omega t} \hat{b}_{L/R\omega}^\dagger + H.c. \quad (2.60)$$

The boundary condition at $x = 0$ follows from Kirchhoff's current law

$$\hat{I}(t) = \frac{\hat{V}_{out}(t) - \hat{V}_{in}(t)}{Z_{tml}} \quad (2.61)$$

where the left hand side $\hat{I}(t) = (C_\kappa/C_r)\dot{\hat{Q}}_r(t)$ is the current ejected by the sample, with \hat{Q}_r the oscillator charge, while the right hand side is the transmission line voltage difference at $x = 0$. A mode expansions of the operators involved in $\hat{I}(t)$ leads to the standard input-output relation

$$\hat{b}_{out}(t) - \hat{b}_{in}(t) = \sqrt{\kappa} \hat{a}(t), \quad (2.62)$$

where the input and output fields are defined as

$$\hat{b}_{in}(t) = \frac{-i}{\sqrt{2\pi}} \int_{-\infty}^{\infty} d\omega \hat{b}_{L\omega} e^{-i(\omega - \omega_r)t}, \quad (2.63)$$

$$\hat{b}_{out}(t) = \frac{-i}{\sqrt{2\pi}} \int_{-\infty}^{\infty} d\omega \hat{b}_{R\omega} e^{-i(\omega - \omega_r)t} \quad (2.64)$$

and satisfy the commutation relations $[\hat{b}_{in}(t), \hat{b}_{in}^\dagger(t')] = [\hat{b}_{out}(t), \hat{b}_{out}^\dagger(t')] = \delta(t - t')$. To arrive at the input-output relation, terms rotating at $\omega + \omega_r$ have been dropped based on the already mentioned assumption that the system only responds to frequencies $\omega \approx \omega_r$ such that these terms are fast rotating. In turn, this approximation allows to extend the range of integration from $(0, \infty)$ to $(-\infty, \infty)$. We have also approximated $\lambda(\omega) \approx \lambda(\omega_r)$ over the relevant frequency range. These approximations are compatible with those used to arrive at the Lindblad-form Markovian master equation of $\dot{\rho} = -i[\hat{H}_S, \rho] + \kappa(\bar{n}_\kappa + 1)\mathcal{D}[\hat{a}]\rho + \kappa\bar{n}_\kappa\mathcal{D}[\hat{a}^\dagger]\rho$.

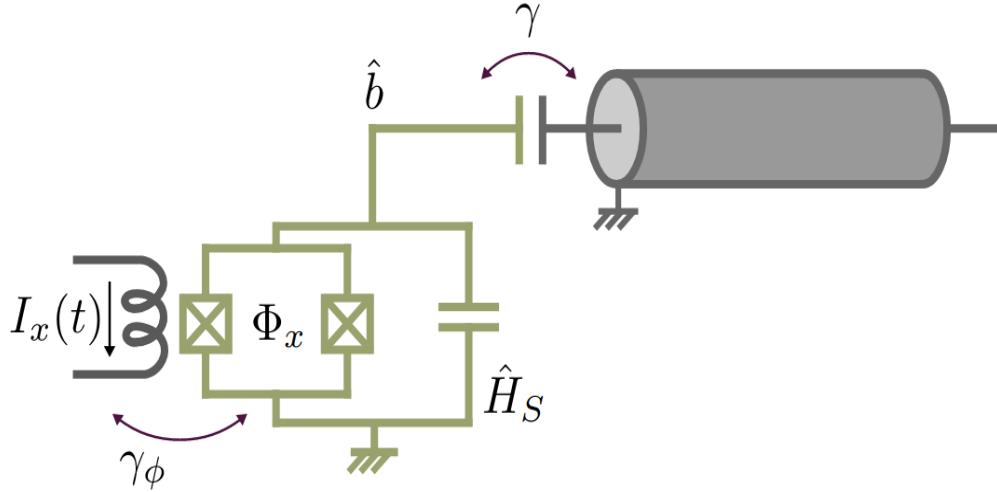
The same expressions and approximations can be used to obtain the equation of motion for the resonator field $\hat{a}(t)$ in the Heisenberg picture, which takes the form

$$\dot{\hat{a}}(t) = i[\hat{H}_S, \hat{a}(t)] - \frac{\kappa}{2}\hat{a}(t) + \sqrt{\kappa}\hat{b}_{in}(t). \quad (2.65)$$

This expression shows that the resonator dynamics is determined by the input field while the output can be found from the input and the system dynamics. The output field thus holds information about the system's response to the input and which can be measured to, indirectly, give us access to information about the dynamics of the system. As will be discussed later on, this can be done, for example, by measuring the voltage at some $x > 0$ away from the oscillator. Under the approximations used above, this voltage can be expressed as

$$\hat{V}(x, t) \approx \sqrt{\frac{\hbar\omega_r Z_{tml}}{2}} [e^{i\omega_r x/v - i\omega_r t} \hat{b}_{out}(t) + e^{i\omega_r x/v - i\omega_r t} \hat{b}_{in}(t) + H.c.]. \quad (2.66)$$

Note that this approximate expression assumes that all relevant frequencies are near ω_r and furthermore neglects all non-Markovian time-delay effects.



2.2.5.3 Qubit Relaxation And Dephasing

The master equation for $\dot{\rho}$ was derived for an oscillator coupled to a transmission line, but this form of the master equation is quite general. In fact,

$$\hat{H} = \hat{H}_S + \hat{H}_{t\text{ml}} - \hbar \int_0^\infty d\omega \lambda(\omega) (\hat{b}_\omega^\dagger - \hat{b}_\omega)(\hat{a}^\dagger - \hat{a})$$

is itself a very generic system-bath Hamiltonian that can be used to model dissipation due to a variety of different noise sources. To model damping of an arbitrary quantum system, for example a transmon qubit or a coupled resonator-transmon system, the operator \hat{a} in this equation is simply replaced with the relevant system operator that couples to the transmission line (or, more generally, the bath).

For the case of transmon in Fig. 2.6, \hat{H}_S in the master equation for $\dot{\rho}$ replaced with the Hamiltonian $\hat{H}_q = \hbar\omega_q \hat{b}^\dagger \hat{b} - \frac{E_C}{2} \hat{b}^\dagger \hat{b}^\dagger \hat{b} \hat{b}$ together with the additional replacements $\mathcal{D}[\hat{a}] \bullet \rightarrow \mathcal{D}[\hat{b}] \bullet$, $\mathcal{D}[\hat{a}^\dagger] \bullet \rightarrow \mathcal{D}[\hat{b}^\dagger] \bullet$, and $\kappa \rightarrow \gamma$.

Here, $\gamma = 2\pi\lambda(\omega_q)^2$ is the relaxation rate of the artificial atom which is related to the qubit-environment coupling strength evaluated at the qubit frequency. This immediately leads to the master equation

$$\dot{\rho} = -i [\hat{H}_q, \rho] + \gamma (\bar{n}_\gamma + 1) \mathcal{D}[\hat{b}] \rho + \gamma \bar{n}_\gamma \mathcal{D}[\hat{b}^\dagger] \rho, \quad (2.67)$$

where ρ now refers to the transmon state and \bar{n}_γ is the thermal photon number of the transmon's environment. It is often assumed that $\bar{n}_\gamma \rightarrow 0$ but, just like for the oscillator, a residual thermal population is often observed in practice.

Superconducting quantum circuits can also suffer from dephasing caused, for example, by fluctuations of parameters controlling their transition frequency and by dispersive coupling to other degrees of freedom in their environment. For a transmon, a phenomenological model

for dephasing can be introduced by adding $2\gamma_\varphi \mathcal{D}[\hat{b}^\dagger \hat{b}] \rho$ term with γ_φ the pure dephasing rate to the master equation. Because of its insensitivity to charge noise (Fig. 2.3), γ_φ is often very small for the 0-1 transition of transmon qubits. Given that charge dispersion increases exponentially with level number, dephasing due to the charge noise can, however, be apparent on higher transmon levels. Another source of dephasing for the transmon is the residual thermal photon population of a resonator to which the transmon is dispersively coupled. This can be understood from the form of the interaction in the dispersive regime, $\chi_{ab} \hat{a}^\dagger \hat{a} \hat{b}^\dagger \hat{b}$, where fluctuations of the photon number lead to fluctuations in the qubit frequency and therefore to dephasing. We note that a term of the form of $2\gamma_\varphi \mathcal{D}[\hat{b}^\dagger \hat{b}] \rho$ but with $\hat{b}^\dagger \hat{b}$ replaced by $\hat{a}^\dagger \hat{a}$ can be added to the master equation of the oscillator to model this aspect. Oscillator dephasing rates are, however, typically small and this contribution is often neglected. Other sources of relaxation and dephasing include two-level systems within the materials and interfaces of the devices, quasiparticles phenomenon including infrared radiation and even ionization radiation.

Combining the above results, the master equation for a transmon subject to relaxation and dephasing assuming $\bar{n}_\gamma \rightarrow 0$ is

$$\dot{\rho} = -i [\hat{H}_q, \rho] + \gamma \mathcal{D}[\hat{b}] \rho + 2\gamma_\varphi \mathcal{D}[\hat{b}^\dagger \hat{b}] \rho \quad (2.68)$$

It is common to express this master equation in the two level approximation of the transmon, something that is obtained simply by taking $\hat{H}_q \rightarrow \hbar\omega_a \hat{\sigma}_z / 2$, $\hat{b}^\dagger \hat{b} \rightarrow (\hat{\sigma}_z + 1) / 2$, $\hat{b} \rightarrow \hat{\sigma}_-$ and $\hat{b}^\dagger \rightarrow \hat{\sigma}_+$.

Note that the rates γ and γ_φ appearing in the above expressions are related to the characteristic T_1 relaxation time and T_2 coherence time of the artificial atom which are defined as

$$T_1 = \frac{1}{\gamma_1} = \frac{1}{\gamma_\downarrow + \gamma_\uparrow} \simeq \frac{1}{\gamma} \quad (2.69)$$

$$T_2 = \frac{1}{\gamma_2} = \left(\frac{\gamma_1}{2} + \gamma_\varphi \right)^{-1}$$

where $\gamma_\downarrow = (\bar{n}_\gamma + 1)\gamma$, $\gamma_\uparrow = \bar{n}_\gamma\gamma$. The approximation for T_1 is for $\bar{n}_\gamma \rightarrow 0$. At zero temperature, T_2 is the characteristic time for the artificial atom to relax from its first excited state to the ground state. On the other hand, T_2 is the dephasing time, which quantifies the characteristic lifetime of coherent superpositions, and includes both a contribution from pure dephasing (γ_φ) and relaxation (γ_1). Current best values for the T_1 and T_2 time of transmon qubits is in the 50 to 120 μ s range for aluminum-based transmons. Relaxation times above 300 μ s have been reported in transmon qubits where the transmon pads have been made with tantalum rather than aluminum, but the Josephson junction still made from aluminum and aluminum oxide. Other superconducting qubits also show large relaxation and coherence times. Examples are $T_1, T_2 \sim 300 \mu$ s for heavy-fluxonium qubits and $T_1 \sim 1.6$ ms and $T_2 \sim 25 \mu$ s for the 0 - π qubit.

Qubit relaxation and incoherent excitation occur due to uncontrolled exchange of GHz frequency photons between the qubit and its environment. These processes are observed to be well described by the Markovian master equation of

$$\dot{\rho} = -i \left[\hat{H}_q, \rho \right] + \gamma \mathcal{D}[\hat{b}] \rho + 2\gamma_\varphi \mathcal{D} \left[\hat{b}^\dagger \hat{b} \right] \rho \quad (2.70)$$

In contrast, the dynamics leading to dephasing are typically non-Markovian, happening at low-frequencies (i.e. slow time scales set by the phase coherence time itself). As a result, these processes cannot very accurately be described by a Markovian master equation such as last equation for $\dot{\rho}$. This equation thus represents a somewhat crude approximation to dephasing in superconducting qubits. That being said, in practice, the Markovian theory is still useful in particular because it correctly predicts the results of experiments probing the steady-state response of the system.

2.2.5.4 Dissipation in the dispersive regime

We now turn to a description of dissipation for the coupled transmon-resonator system of Sec. Light Matter Interaction in Circuit QED. Assuming that the transmon and the resonator are coupled to independent baths as illustrated in 2.7, the master equation for this composite system is (taking $\bar{n}_{\kappa,\gamma} \rightarrow 0$ for simplicity),

$$\begin{aligned} \dot{\rho} = & -i[\hat{H}, \rho] + \kappa \mathcal{D}[\hat{a}] \rho + \gamma \mathcal{D}[\hat{b}] \rho \\ & + 2\gamma_\varphi \mathcal{D} \left[\hat{b}^\dagger \hat{b} \right] \rho \end{aligned} \quad (2.71)$$

where ρ is now a density matrix for the total system, and \hat{H} describes the coupled systems as in equation

$$\begin{aligned} \hat{H} \approx & \hbar\omega_r \hat{a}^\dagger \hat{a} + \hbar\omega_q \hat{b}^\dagger \hat{b} - \frac{E_C}{2} \hat{b}^\dagger \hat{b}^\dagger \hat{b} \hat{b} \\ & + \hbar g \left(\hat{b}^\dagger \hat{a} + \hat{b} \hat{a}^\dagger \right). \end{aligned} \quad (2.72)$$

Importantly, the above expression is only valid at small values of $g/(\omega_r, \omega_q)$. This is because energy decay occurs via transitions between system eigenstates while the above expression describes transitions between the uncoupled bare states. A derivation of the master equation valid at arbitrary g can be found. More important to the present discussion is the fact that, at first glance of the master equation for this composite system gives the impression that dissipative processes influence the transmon and the resonator in completely independent manners. However, because \hat{H} entangles the two systems, the loss, for example, of a resonator photon can lead to qubit relaxation. Moving to the dispersive regime, a more complete picture of dissipation emerges after applying the unitary transformation

$$\hat{U}_{\text{disp}} = \exp \left[\Lambda \left(\hat{a}^\dagger \hat{b} - \hat{a} \hat{b}^\dagger \right) \right] \quad (2.73)$$

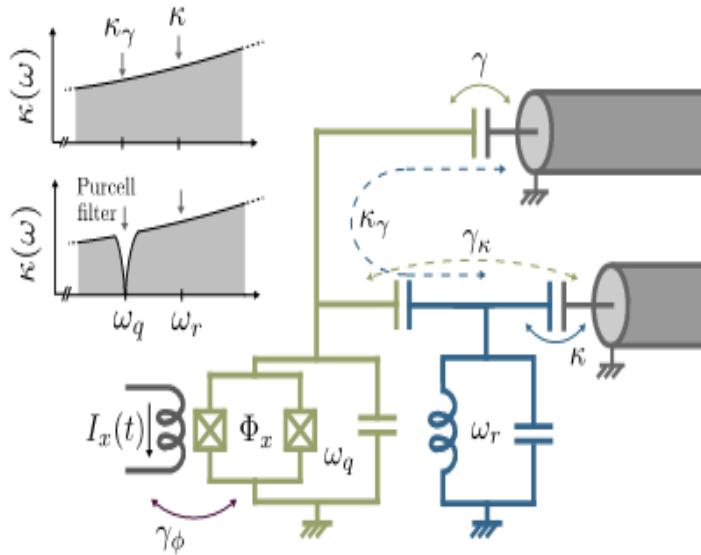
not only on the Hamiltonian but also on the above master equation. Neglecting fast rotating terms and considering corrections to second order in λ (which is consistent if $\kappa, \gamma, \gamma_\varphi = \mathcal{O}(E_C g^2 / \Delta^2)$), leads to dispersive master equation.

$$\begin{aligned}\dot{\rho}_{\text{disp}} = & -i \left[\hat{H}_{\text{disp}}, \rho_{\text{disp}} \right] + (\kappa + \kappa_\gamma) \mathcal{D}[\hat{a}] \rho_{\text{disp}} \\ & + (\gamma + \gamma_\kappa) \mathcal{D}[\hat{b}] \rho_{\text{disp}} 2\gamma_\varphi \mathcal{D}[\hat{b}^\dagger \hat{b}] \rho_{\text{disp}} \\ & + \gamma_\Delta \mathcal{D}[\hat{a}^\dagger \hat{b}] \rho_{\text{disp}} + \gamma_\Delta \mathcal{D}[\hat{b}^\dagger \hat{a}] \rho_{\text{disp}}\end{aligned}\quad (2.74)$$

where,

$$\gamma_\kappa = \left(\frac{g}{\Delta} \right)^2 \kappa, \kappa_\gamma = \left(\frac{g}{\Delta} \right)^2 \gamma, \gamma_\Delta = 2 \left(\frac{g}{\Delta} \right)^2 \gamma_\varphi \quad (2.75)$$

and $\rho_{\text{disp}} = \hat{U}_{\text{disp}}^\dagger \rho \hat{U}_{\text{disp}}$ is the density matrix in the dispersive frame.



This expression has three new rates, the first of which is known as the Purcell decay rate γ_κ . This rate captures the fact that the qubit can relax by emission of a resonator photon. It can be understood simply following equation

$$\begin{aligned}|\overline{g}, \overline{n}\rangle &= \cos(\theta_n/2) |g, n\rangle - \sin(\theta_n/2) |e, n-1\rangle \\ |e, n-1\rangle &= \sin(\theta_n/2) |g, n\rangle + \cos(\theta_n/2) |e, n-1\rangle\end{aligned}\quad (2.76)$$

from the form of the dressed eigenstate $|\overline{e}, \overline{0}\rangle \sim |e, 0\rangle + (g/\Delta)|g, 1\rangle$ which is closest to a qubit excitation $|e\rangle$. This state is the superposition of the qubit first excited state with no photon and, with probability $(g/\Delta)^2$, the qubit ground state with a photon in the resonator. The latter component can decay at the rate κ taking the dressed excited qubit to the ground state $|g, 0\rangle$ with a rate γ_κ . A similar intuition also applies to κ_γ , now associated with a resonator photon loss through a qubit decay event.

The situation is more subtle for the $\gamma_\Delta \mathcal{D} [\hat{a}^\dagger \hat{b}] \rho_{\text{disp}} + \gamma_\Delta \mathcal{D} [\hat{b}^\dagger \hat{a}] \rho_{\text{disp}}$ part of the dispersive master equation. An effective master equation for the transmon only can be obtained from

$$\begin{aligned}\dot{\rho}_{\text{disp}} = & -i \left[\hat{H}_{\text{disp}}, \rho_{\text{disp}} \right] + (\kappa + \kappa_\gamma) \mathcal{D}[\hat{a}] \rho_{\text{disp}} \\ & + (\gamma + \gamma_\kappa) \mathcal{D}[\hat{b}] \rho_{\text{disp}} 2\gamma_\varphi \mathcal{D}[\hat{b}^\dagger \hat{b}] \rho_{\text{disp}} \\ & + \gamma_\Delta \mathcal{D}[\hat{a}^\dagger \hat{b}] \rho_{\text{disp}} + \gamma_\Delta \mathcal{D}[\hat{b}^\dagger \hat{a}] \rho_{\text{disp}}\end{aligned}\quad (2.77)$$

by approximately eliminating the resonator degrees of freedom. This results in the transmon relaxation and excitation rates given approximately by $\bar{n}\gamma_\Delta$, with \bar{n} the average photon number in the resonator. commonly known as dressed-dephasing, this leads to spurious transitions during qubit measurement and can be interpreted as originating from dephasing noise at the detuning frequency Δ that is up- or down-converted by readout photons to cause spurious qubit state transitions.

Because we have taken the shortcut of applying the dispersive transformation on the master equation, the above discussion neglects the frequency dependence of the various decay rates. In a more careful derivation, the dispersive transformation is applied on the system plus bath Hamiltonian, and only then is the master equation derived. The result has the same form as above equation, but with different expressions for the rates. Indeed, it is useful to write $\kappa = \kappa(\omega_r)$ and $\gamma = \gamma(\omega_q)$ to recognize that, while photon relaxation is probing the environment at the resonator frequency ω_r , qubit relaxation is probing the environment at ω_q . With this notation, the first two rates of

$$\gamma_\kappa = \left(\frac{g}{\Delta} \right)^2 \kappa, \kappa_\gamma = \left(\frac{g}{\Delta} \right)^2 \gamma, \gamma_\Delta = 2 \left(\frac{g}{\Delta} \right)^2 \gamma_\varphi \quad (2.78)$$

become in the more careful derivation $\gamma_\kappa = (g/\Delta)^2 \kappa(\omega_q)$ and $\kappa_\gamma = (g/\Delta)^2 \gamma(\omega_r)$. In other words, Purcell decay occurs by emitting a photon at the qubit frequency and not at the resonator frequency as suggested by the completely white noise model used to derive

$$\gamma_\kappa = \left(\frac{g}{\Delta} \right)^2 \kappa, \kappa_\gamma = \left(\frac{g}{\Delta} \right)^2 \gamma, \gamma_\Delta = 2 \left(\frac{g}{\Delta} \right)^2 \gamma_\varphi \quad (2.79)$$

In the same way, it is useful to write the dephasing rate as $\gamma_\varphi = \lambda_\varphi \rightarrow 0$ to recognize the importance of low-frequency noise. Using this notation, the rates in the last two terms of

$$\begin{aligned}\dot{\rho}_{\text{disp}} = & -i \left[\hat{H}_{\text{disp}}, \rho_{\text{disp}} \right] + (\kappa + \kappa_\gamma) \mathcal{D}[\hat{a}] \rho_{\text{disp}} \\ & + (\gamma + \gamma_\kappa) \mathcal{D}[\hat{b}] \rho_{\text{disp}} 2\gamma_\varphi \mathcal{D}[\hat{b}^\dagger \hat{b}] \rho_{\text{disp}} \\ & + \gamma_\Delta \mathcal{D}[\hat{a}^\dagger \hat{b}] \rho_{\text{disp}} + \gamma_\Delta \mathcal{D}[\hat{b}^\dagger \hat{a}] \rho_{\text{disp}}\end{aligned}\quad (2.80)$$

become, respectively, $\gamma_{\Delta} = 2(g/\Delta)^2\gamma_{\varphi}(\Delta)$ and $\gamma_{-\Delta} = 2(g/\Delta)^2\gamma_{\varphi}(-\Delta)$. In short, dressed dephasing probes the noise responsible for dephasing at the transmon-resonator detuning frequency Δ . This observation was used to probe this noise at GHz frequencies.

It is important to note that the observations in this section result from the qubit-oscillator dressing that occurs under the Jaynes-Cummings Hamiltonian. For this reason, the situation is very different if the electric-dipole interaction leading to the Jaynes-Cummings Hamiltonian is replaced by a longitudinal interaction of the form of $\hat{H}_z = \hbar\omega_r \hat{a}^\dagger \hat{a} + \frac{\hbar\omega_q}{2} \hat{\sigma}z + \hbar g z (\hat{a}^\dagger + \hat{a}) \hat{\sigma}_z$. In this case, there is no light-matter dressing and consequently no Purcell decay or dressed-dephasing. This is one of the advantages of this alternative light-matter coupling.

2.2.5.5 Multi-mode Purcell effect and Purcell filters

Up to now we have considered dissipation for a qubit dispersively coupled to a single-mode oscillator. Replacing the latter with a multi-mode resonator leads to dressing of the qubit by all of the resonator modes and therefore to a modification of the Purcell decay rate. Following the above discussion, one may then expect the contributions to add up, leading to the modified rate $\sum_{m=0}^{\infty} (g_m/\Delta_m)^2 \kappa_m$, with m the mode index.

However, when accounting for the frequency dependence of κ_m , g_m and Δ_m , this expression diverges. It is possible to cure this problem using a more refined model including the finite size of the transmon and the frequency dependence of the impedance of the resonator's input and output capacitors.

Given that damping rates in quantum electrical circuits are set by classical system parameter, a simpler approach to compute the Purcell rate exists. It can indeed be shown that $\gamma_\kappa = \text{Re}[Y(\omega_q)]/C_\Sigma$, with $Y(\omega) = 1/Z(\omega)$ the admittance of the electromagnetic environment seen by the transmon. This expression again makes it clear that relaxation probes the environment (here represented by the admittance) at the system frequency. It also suggests that engineering the admittance $Y(\omega)$ such that it is purely reactive at ω_q can cancel Purcell decay (Fig. 2.7). This can be done, for example, by adding a transmission-line stub of appropriate length and terminated in an open circuit at the output of the resonator, something which is known as a Purcell filter. Because of the increased freedom in optimizing the system parameters (essentially decoupling the choice of κ from the qubit relaxation rate), various types of Purcell filters are commonly used experimentally.

2.2.5.6 Controlling quantum systems with microwave drives

While connecting a quantum system to external transmission lines leads to losses, such connections are nevertheless necessary to control and measure the system. Consider a continuous microwave tone of frequency ω_d and phase ϕ_d applied to the input port of the resonator. A simple approach to model this drive is based on the input-output approach of section input-output theory in electrical networks. Indeed, the drive can be taken into account by replacing the input field $\hat{b}(t)$ in equation

$$\dot{\hat{a}}(t) = i [\hat{H}_S, \hat{a}(t)] - \frac{\kappa}{2} \hat{a}(t) + \sqrt{\kappa} \hat{b}_{\text{in}}(t) \quad (2.81)$$

with $\hat{b}_{\text{in}}(t) \rightarrow \hat{b}_{\text{in}}(t) + \beta(t)$ where

$$\beta(t) = A(t) \exp(-i\omega_d t - i\phi_d) \quad (2.82)$$

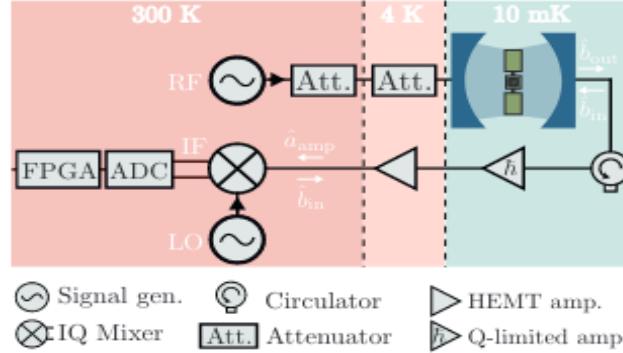
is the coherent classical part of the input field of amplitude $A(t)$.

The resulting term $\sqrt{\kappa}\beta(t)$ in the Langevin equation can be absorbed in the system Hamiltonian with the replacement $\hat{H}_S \rightarrow \hat{H}_S + \hat{H}_d$ where

$$\hat{H}_d = \hbar [\varepsilon(t) \hat{a}^\dagger e^{-i\omega_d t - i\phi_d} + \varepsilon^*(t) \hat{a} e^{i\omega_d t + i\phi_d}] \quad (2.83)$$

with $\varepsilon(t) = i\sqrt{\kappa}A(t)$

the possibly time-dependent amplitude of the drive as seen by the resonator mode. Generalizing to multiple drives on the resonator and/or drives on the transmon is straightforward.



Moreover, the Hamiltonian \hat{H}_d is the generator of displacement in phase space of the resonator. As a result, by choosing appropriate parameters for the drive, evolution under \hat{H}_d will bring the intra-resonator state from vacuum to an arbitrary coherent state.

$$|\alpha\rangle = \hat{D}(\alpha)|0\rangle = e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \quad (2.84)$$

,

where $\hat{D}(\alpha)$ is known as the displacement operator and takes the form $\hat{D}(\alpha) = e^{\alpha\hat{a}^\dagger - \alpha^*\hat{a}}$.

As discussed in the next section, coherent states play an important role in qubit readout in circuit QED.

It is important to note that \hat{H}_d derives from following equation,

$$\dot{\hat{a}}(t) = i [\hat{H}_S, \hat{a}(t)] - \frac{\kappa}{2} \hat{a}(t) + \sqrt{\kappa} \hat{b}_{\text{in}}(t) \quad (2.85)$$

with

$$\hat{b}_{\text{in}}(t) \rightarrow \hat{b}_{\text{in}}(t) + \beta(t)$$

which is itself the result of a rotating-wave approximation. As can be understood from

$$\begin{aligned} \hat{H}_T &= \frac{(\hat{Q} - Q_g)^2}{2C_\Sigma} - E_J \cos\left(\frac{2\pi}{\Phi_0}\hat{\Phi}\right) \\ &= 4E_C (\hat{n} - n_g)^2 - E_J \cos\hat{\varphi}. \end{aligned} \quad (2.86)$$

before this approximation, the drive rather takes the form $i\hbar\varepsilon(t) \cos(\omega_d t + \phi_d) (\hat{a}^\dagger - \hat{a})$. Although \hat{H}_d is sufficient in most cases of practical interest,

departures from the predictions of equation

$$\hat{H}_d = \hbar[\varepsilon(t)\hat{a}^\dagger e^{-i\omega_d t - i\phi_d} + \varepsilon^*(t)\hat{a} e^{i\omega_d t + i\phi_d}] \quad (2.87)$$

can be seen at large drive amplitudes.

2.2.6 Measurements in Circuit QED

Before the development of circuit QED, the quantum state of superconducting qubits was measured by fabricating and operating a measurement device, such as a single-electron transistor, in close proximity to the qubit. A challenge with such an approach is that the readout circuitry must be strongly coupled to the qubit during measurement so as to extract information on a time scale much smaller than T_1 , while being well decoupled from the qubit when the measurement is turned off to avoid unwanted back-action. Especially given that measurement necessarily involves dissipation, simultaneously satisfying these two requirements is challenging. Circuit QED, however, has several advantages to offer over the previous approaches. Indeed, as discussed in further detail in this section, qubit readout in this architecture is realized by measuring scattering of a probe tone off an oscillator coupled to the qubit. This approach first leads to an excellent measurement on/off ratio since qubit readout only occurs in the presence of the probe tone. A second advantage is that the necessary dissipation now occurs away from the qubit, essentially at a voltage meter located at room temperature, rather than in a device fabricated in close proximity to the qubit. Unwanted energy exchange is moreover inhibited when working in the dispersive regime where the effective qubit-resonator interaction

$$\hat{H}_{\text{disp}} \approx \hbar\omega'_r \hat{a}^\dagger \hat{a} + \frac{\hbar\omega'_q}{2} \hat{\sigma}_z + \hbar\chi \hat{a}^\dagger \hat{a} \hat{\sigma}_z, \quad (2.88)$$

is such that even the probe-tone photons are not absorbed by the qubit. As a result, the backaction on the qubit is to a large extent limited to the essential dephasing that quantum measurements must impart on the measured system leading, in principle, to a quantum non-demolition (QND) qubit readout. Because of the small energy of microwave photons with respect to optical photons, single-photon detectors in the microwave frequency regime are still being developed. Therefore, measurements in circuit QED rely on amplification of weak microwave signals followed by detection of field quadratures using heterodyne detection. Before discussing qubit readout, the objective of the next subsection is to explain these terms and go over the main challenges related to such measurements in the quantum regime.

2.2.6.1 Microwave field detection

The signal of a microwave source is directed to the input port of the resonator first going through a series of attenuators thermally anchored at different stages of the dilution refrigerator. The role of these attenuators is to absorb the room-temperature thermal noise propagating towards the sample. The field transmitted by the resonator is first amplified, then mixed with a reference signal, converted from analog to digital, and finally processed

with an FPGA or recorded. Circulators are inserted before the amplification stage to prevent noise generated by the amplifier from reaching the resonator. Circulators are directional devices that transmit signals in the forward direction while strongly attenuating signals propagating in the reverse direction. In practice, circulators are bulky off-chip devices relying on permanent magnets that are not compatible with the requirement for integration with superconducting quantum circuits. They also introduce additional losses, for example due to insertion losses and off-chip cable losses. Significant effort is currently being devoted to developing compact, on-chip, superconducting circuit-based circulators.

In practice, the different components and cables of the measurement chain have a finite bandwidth which we will assume to be larger than the bandwidth of the signal of interest $b_{\text{out}}(t)$ at the output of the resonator. To account for the finite bandwidth of the measurement chain and to simplify the following discussion, it is useful to consider the filtered output field

$$\begin{aligned}\hat{a}_f(t) &= \left(f * \hat{b}_{\text{out}} \right)(t) \\ &= \int_{-\infty}^{\infty} d\tau f(t - \tau) \hat{b}_{\text{out}}(\tau) \\ &= \int_{-\infty}^{\infty} d\tau f(t - \tau) \left[\sqrt{\kappa} \hat{a}(\tau) + \hat{b}_{\text{in}}(\tau) \right]\end{aligned}\quad (2.89)$$

which is linked to the intra-cavity field \hat{a} via the input-output boundary condition $\hat{b}_{\text{out}}(t) - \hat{b}_{\text{in}}(t) = \sqrt{\kappa} \hat{a}(t)$ which we have used in the last line. In this expression, the filter function $f(t)$ is normalized to $\int_{-\infty}^{\infty} dt |f(t)|^2 = 1$ such that $[\hat{a}_f(t), \hat{a}_f^\dagger(t)] = 1$. As will be discussed later in the context of qubit readout, in addition to representing the measurement bandwidth, filter functions are used to optimize the distinguishability between the qubit states.

Ignoring the presence of the circulator and assuming that a phase-preserving amplifier (i.e. an amplifier that amplifies both signal quadratures equally) is used, in the first stage of the measurement chain the signal is transformed according to

$$\hat{a}_{\text{amp}} = \sqrt{G} \hat{a}_f + \sqrt{G - 1} \hat{h}^\dagger \quad (2.90)$$

where G is the power gain and \hat{h}^\dagger accounts for noise added by the amplifier. The presence of this added noise is required for amplified signal to obey the bosonic commutation relation, $[\hat{a}_{\text{amp}}, \hat{a}_{\text{amp}}^\dagger] = 1$. Equivalently, the noise must be present because the two quadratures of the signal are canonically conjugate. amplification of both quadratures without added noise would allow us to violate the Heisenberg uncertainty relation between the two quadratures.

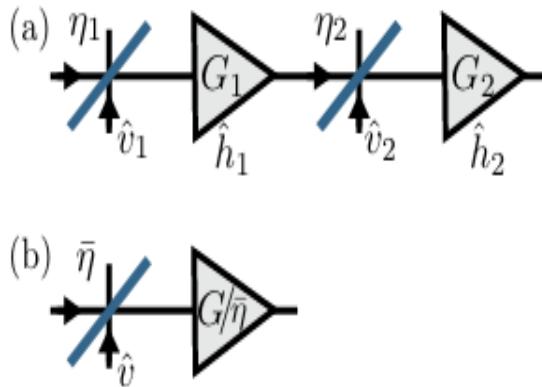
In a standard parametric amplifiers, \hat{a}_f in last equation represents the amplitude of the signal mode and h represents the amplitude of a second mode called idler. The physical interpretation of last equation is that an ideal amplifier performs a Bogoljubov transformation on the signal and idler modes. The signal mode is amplified, but the requirement that

the transformation be canonical implies that the (phase conjugated and amplified) quantum noise from the idler port must appear in the signal output port. Ideally, the input to the idler is vacuum with $\langle \hat{h}^\dagger \hat{h} \rangle = 0$ and $\langle \hat{h} \hat{h} \rangle = 1$, so the amplifier only adds quantum noise. Near quantum-limited amplifiers with ~ 20 dB power gain approaching this ideal behaviour are now routinely used in QED experiments. These Josephson junction-based devices, as well as the distinction between phase-preserving and phase-sensitive amplification will be discussed in further section.

To measure the very weak signals that are typical in circuit QED, the output of the first near-quantum limited amplifier is further amplified by a low-noise high-electron-mobility transistor (HEMT) amplifier. The latter acts on the signal again following

$$\hat{a}_{amp} = \sqrt{G} \hat{a}_f + \sqrt{G - 1} \hat{h}^\dagger$$

, now with a larger power gain $\sim 30 - 40$ dB but also larger added noise photon number. The very best cryogenic HEMT amplifiers in the 4-8 GHz band have noise figures as low as $\langle \hat{h}^\dagger \hat{h} \rangle \sim 5-10$. However, the effect of attenuation due to cabling up to the previous element of the amplification chain, i.e. quantum-limited amplifier or the sample of interest itself, can degrade this figure significantly. A more complete understanding of the added noise in this situation can be derived from figure below.



There, beam splitters of transmittivity $\eta_{1,2}$ model the attenuation leading to the two amplifiers of gain labelled G_1 and G_2 . Taking into account vacuum noise $\hat{v}_{1,2}$ at the beam splitters, the input-output expression of this chain can be cast under the form of

$$\hat{a}_{amp} = \sqrt{G} \hat{a}_f + \sqrt{G - 1}$$

with a total gain $G_T = \eta_1 \eta_2 G_1 G_2$ and noise mode \hat{h}_T^\dagger corresponding to the total added noise number

$$\begin{aligned} N_T &= \frac{1}{G_T - 1} [\eta_1 (G_1 - 1) G_2 (N_1 + 1) \\ &\quad + (G_2 - 1) (N_2 + 1)] - 1 \\ &\approx \frac{1}{\eta_1} \left[1 + N_1 + \frac{N_2}{\eta_2 G_1} \right] - 1 \end{aligned} \tag{2.91}$$

with $N_i = \langle \hat{h}_i^\dagger \hat{h}_i \rangle$ with $i = 1, 2, T$. If the gain G_1 of the first amplifier is large, the noise of the chain is dominant by the noise N_1 of the first amplifier. This emphasizes the importance of using near quantum-limited amplifiers with low noise in the first stage of the chain. In the literature, the quantum efficiency $\eta = 1/(N_T + 1)$ is often used to characterize the measurement chain, with $\eta = 1$ in the ideal case $N_T = 0$.

It is worthwhile to note that another definition of the quantum efficiency can often be found in the literature. This alternative definition based on 2.9(b) where a noisy amplifier of gain G is replaced by a noiseless amplifier of gain $G/\bar{\eta}$ preceded by a fictitious beam splitter of transmittivity $\bar{\eta}$ adding vacuum noise to the amplifier's input. The quantum efficiency corresponds, here, to the transmittivity $\bar{\eta}$ of the fictitious beam splitter. The input-output relation of the network of 2.9(b) with its noiseless phase-preserving amplifier reads

$$\hat{a}_{\text{amp}} = \sqrt{G/\bar{\eta}} \left(\sqrt{\bar{\eta}} \hat{a}_f + \sqrt{1 - \bar{\eta}} \hat{v} \right)$$

, something which can be expressed as

$$\langle |\hat{a}_{\text{amp}}|^2 \rangle = \frac{G}{\bar{\eta}} \left[(1 - \bar{\eta}) \frac{1}{2} + \bar{\eta} \langle |\hat{a}_f|^2 \rangle \right] \quad (2.92)$$

with $\langle |\hat{O}|^2 \rangle = \langle \{\hat{O}^\dagger, \hat{O}\} \rangle / 2$ the symmetrized fluctuations. The first term of the above expression corresponds to the noise added by the amplifier, here represented by vacuum noise added to the signal before amplification, while the second term corresponds to noise in the signal at the input of the amplifier. On the other hand, Eq. 2.90 for a noisy amplifier can also be cast in the form of Eq. 2.92 with

$$\langle |\hat{a}_{\text{amp}}|^2 \rangle = G (\mathcal{A} + \langle |\hat{a}_f|^2 \rangle), \quad (2.93)$$

where we have introduced the added noise

$$\mathcal{A} = \frac{(G - 1)}{G} \left(\langle \hat{h}^\dagger \hat{h} \rangle + \frac{1}{2} \right). \quad (2.94)$$

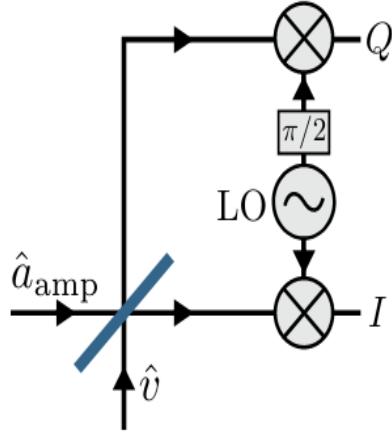
In the limit of low amplifier noise $\langle \hat{h}^\dagger \hat{h} \rangle \rightarrow 0$ and large gain, the added noise is found to be bounded by $\mathcal{A} \geq (1 - G^{-1}) / 2 \simeq 1/2$ corresponding to half a photon of noise. Using Eqs. 2.92 and 2.93, the quantum efficiency of a phase-preserving amplifier can therefore be written as $\bar{\eta} = 1/(2\mathcal{A} + 1) \leq 1/2$ and is found to be bounded by 1/2 in the ideal case. Importantly, the concept of quantum efficiency is not limited to amplification, and can be applied to the whole measurement chain illustrated in Fig. 2.8.

Using Eqs. 2.66 and 2.90, the voltage after amplification can be expressed as

$$\hat{V}_{\text{amp}}(t) \simeq \sqrt{\frac{\hbar\omega_{\text{RF}}Z_{\text{tm}}}{2}} [e^{-i\omega_{\text{RF}}t} \hat{a}_{\text{amp}} + \text{H.c.}] \quad (2.95)$$

where ω_{RF} is the signal frequency. To simplify the expressions, we have dropped the phase associated to the finite cable length. We have also dropped the contribution from the input

field $\hat{b}_{in}(t)$ moving towards the amplifier in the opposite direction at this point, because this field is not amplified and therefore gives a very small contribution compared to the amplified output field. Recall, however, the contribution of this field to the filtered signal Eq. 2.2.6.1



Different strategies can be used to extract information from the amplified signal, and here we take the next stage of the chain to be an IQ-mixer. As schematically illustrated in 2.10, in this microwave device the signal first encounters a power divider, illustrated here as a beam splitter to account for added noise due to internal modes, followed in each branch by mixers with local oscillators (LO) that are offset in phase by $\pi/2$. The LO consists in a reference signal of well-defined amplitude A_{LO} , frequency ω_{LO} and phase ϕ_{LO} :

$$V_{LO}(t) = A_{LO} \cos(\omega_{LO} t - \phi_{LO}) \quad (2.96)$$

Mixers use nonlinearity to down-convert the input signal to a lower frequency referred to as the intermediate frequency (IF) signal. Describing first the signal as a classical voltage $V_{RF}(t) = A_{RF} \cos(\omega_{RF} t + \phi_{RF})$, the output at one of these mixers is

$$\begin{aligned} V_{\text{mixer}}(t) &= KV_{RF}(t)V_{LO}(t) \\ &= \frac{1}{2}KA_{LO}A_{RF}\{\cos[(\omega_{LO} - \omega_{RF})t - \phi_{LO}] \\ &\quad + \cos[(\omega_{LO} + \omega_{RF})t - \phi_{LO}]\} \end{aligned} \quad (2.97)$$

where K accounts for voltage conversion losses. According to the above expression, mixing LO results in two sidebands at frequencies $\omega_{LO} \pm \omega_{RF}$. The high frequency component is filtered out with a low-pass filter leaving (not shown) only the lower sideband of frequency $\omega_{IF} = \omega_{LO} - \omega_{RF}$. The choice $\omega_{IF} \neq 0$ is known as heterodyne detection. Taking the LO frequency such that ω_{IF} is in the range of few tens to a few hundreds of MHz, the signal can be digitized using an analog to digital converter (ADC) with a sampling rate chosen in accordance with the bandwidth set by the choice of IF frequency and the signal bandwidth, typically a few MHz to a few ten of MHz if set by the bandwidth $\kappa/2\pi$ of the cavity chosen for the specific circuit QED application such as qubit readout. The recorded signal can then

be averaged, or analyzed in more complex ways, using real-time field-programmable gate array (FPGA) electronics and processed offline.

Going back to a quantum mechanical description of the signal by combining Eqs. 2.95 and 2.97, the IF signals at the I and Q ports of the IQ-mixer read

$$\begin{aligned}\hat{V}_I(t) &= V_{\text{IF}} \left[\hat{X}_f(t) \cos(\omega_{\text{IFT}}t) - \hat{P}_f(t) \sin(\omega_{\text{IFT}}t) \right] \\ &\quad + \hat{V}_{\text{noise}, I}(t) \\ \hat{V}_Q(t) &= -V_{\text{IF}} \left[\hat{P}_f(t) \cos(\omega_{\text{IFT}}t) + \hat{X}_f(t) \sin(\omega_{\text{IFT}}t) \right] \\ &\quad + \hat{V}_{\text{noise}, Q}(t)\end{aligned}\tag{2.98}$$

where we have taken $\phi_{\text{LO}} = 0$ in the I arm of the IQ mixer, and $\phi_{\text{LO}} = \pi/2$ in the Q arm. We have defined $V_{\text{IF}} = KA_{\text{LO}}\sqrt{\kappa G Z_{\text{tm1}} \hbar \omega_{\text{RF}}/2}$, and $\hat{V}_{\text{noise}, I/Q}$ as the contributions from the amplifier noise and any other added noise. We have also introduced the quadratures

$$\hat{X}_f = \frac{\hat{a}_f^\dagger + \hat{a}_f}{2}, \quad \hat{P}_f = \frac{i(\hat{a}_f^\dagger - \hat{a}_f)}{2},\tag{2.99}$$

the dimensionless position and momentum operators of the simple harmonic oscillator, here defined such that $[\hat{X}_f, \hat{P}_f] = i/2$. Taken together, $\hat{V}_I(t)$ and $\hat{V}_Q(t)$ trace a circle in the $x_f - p_f$ plane and contain information about two quadratures at all times. It is therefore possible to digitally transform the signals by going to a frame where they are stationary using the rotation matrix

$$R(t) = \begin{pmatrix} \cos(\omega_{\text{IFT}}t) & -\sin(\omega_{\text{IFT}}t) \\ \sin(\omega_{\text{IFT}}t) & \cos(\omega_{\text{IFT}}t) \end{pmatrix}\tag{2.100}$$

to extract $\hat{X}_f(t)$ and $\hat{P}_f(t)$. We note that the case $\omega_{\text{IF}} = 0$ is generally known as homodyne detection. In this situation, the IF signal in one of the arms of the IQ-mixer is proportional to

$$\begin{aligned}\hat{X}_{f,\phi_{\text{LO}}} &= \frac{\hat{a}_f^\dagger e^{i\phi_{\text{LO}}} + \hat{a}_f e^{-i\phi_{\text{LO}}}}{2} \\ &= \hat{X}_f \cos \phi_{\text{LO}} + \hat{P}_f \sin \phi_{\text{LO}}\end{aligned}\tag{2.101}$$

While this is in appearance simpler and therefore advantageous, this approach is susceptible to $1/f$ noise and drift because the homodyne signal is at DC. It is also worthwhile to note that homodyne detection as realized with the approach described here differs from optical homodyne detection which can be performed in a noiseless fashion (in the present case, noise is added at the very least by the phase-preserving amplifiers and the noise port of the IQ mixer).

2.2.6.2 Phase-space representations and their relation to field detection

In the context of field detection, it is particularly useful to represent the quantum state of the electromagnetic field using phase-space representations and here we focus on Wigner function

and the Husimi-Q distribution. This discussion applies equally well to the intra-cavity field \hat{a} as to the filtered output field \hat{a}_f .

The Wigner function is a quasiprobability distribution given by the Fourier transform

$$W_p(x, p) = \frac{1}{\pi^2} \iint_{-\infty}^{\infty} dx' dp' C_p(x', p') e^{2i(px' - xp')} \quad (2.102)$$

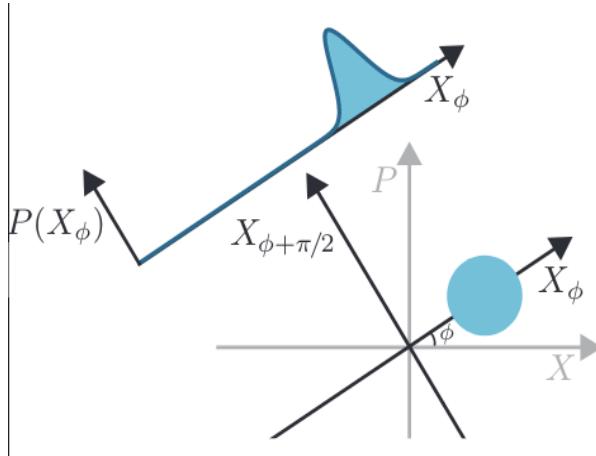
of the characteristic function

$$C_\rho(x, p) = \text{Tr} \left\{ \rho e^{2i(p\hat{X} - x\hat{P})} \right\} \quad (2.103)$$

With ρ the state of the electromagnetic field, $C_\rho(x, p)$ can be understood as the expectation value of the displacement operator

$$\hat{D}(\alpha) = e^{2i(p\hat{X} - x\hat{P})} = e^{\alpha\hat{a}^\dagger - \alpha^*\hat{a}} \quad (2.104)$$

with $\alpha = x + ip$.



Coherent states, already introduced in Eq. 2.84, have particularly simple Wigner functions. Indeed, as illustrated schematically in 2.11, the Wigner function $W_{|\beta\rangle}(\alpha)$ of the coherent state $|\beta\rangle$ is simply a Gaussian centered at β in phase space:

$$W_{|\beta\rangle}(\alpha) = \frac{2}{\pi} e^{-2|\alpha - \beta|^2} \quad (2.105)$$

The width $1/(\sqrt{2})$ of the Gaussian is a signature of quantum noise and implies that coherent states saturate the Heisenberg inequality, $\Delta X \Delta P = 1/2$ with $\Delta \mathcal{O}^2 = \langle \hat{\mathcal{O}}^2 \rangle - \langle \hat{\mathcal{O}} \rangle^2$. We note that, in contrast to Eq. 2.105, Wigner functions take negative values for non-classical states of the field.

In the context of dispersive qubit measurements, the Wigner function is particularly useful because it is related to the probability distribution for the outcome of the measurements of

quadratures \hat{X} and \hat{P} . Indeed, the marginals $P(x)$ and $P(p)$, obtained by integrating $W_{\rho(x,p)}$ along the orthogonal quadrature, are simply given by

$$\begin{aligned} P(x) &= \int_{-\infty}^{\infty} dp W_{\rho}(x, p) = \langle x | \rho | x \rangle \\ P(p) &= \int_{-\infty}^{\infty} dx W_{\rho}(x, p) = \langle p | \rho | p \rangle \end{aligned} \quad (2.106)$$

where $|x\rangle$ and $|p\rangle$ are the eigenstate of \hat{X} and \hat{P} , respectively. This immediately implies that the probability distribution of the outcomes of an ideal homodyne measurement of the quadrature \hat{X}_{ϕ} is given by $P(x_{\phi})$ obtained by integrating the Wigner function $W_{\rho}(\alpha)$ along the orthogonal quadrature $\hat{X}_{\phi+\pi/2}$. This is schematically illustrated for a coherent state in Fig. 2.11. Another useful phase-space function is the Husimi-Q distribution which, for a state ρ , takes the simple form

$$Q_{\rho}(\alpha) = \frac{1}{\pi} \langle \alpha | \rho | \alpha \rangle \quad (2.107)$$

This function represents the probability distribution of finding ρ in the coherent state $|\alpha\rangle$ and, in contrast to $W_{\rho}(\alpha)$, it is therefore always positive. Since $Q_{\rho}(\alpha)$ and $W_{\rho}(\alpha)$ are both complete description of the state ρ , it is not surprising that one can be expressed in terms of the other. for example, in terms of Wigner function, the Q-function takes the form

$$Q_{\rho}(\alpha) = \frac{2}{\pi} \int_{-\infty}^{\infty} d^2 \beta W_{\rho}(\beta) e^{-2|\alpha-\beta|^2} = W_{\rho}(\alpha) * W_{|0\rangle}(\alpha) \quad (2.108)$$

The Husimi-Q distribution $Q_{\rho}(\alpha)$ is thus obtained by convolution of the Wigner function with a Gaussian, and therefore smoother than $W_{\rho}(\alpha)$. As made clear by the second equality, this Gaussian is in fact that the Wigner function of the vacuum state, $W_{|0\rangle}(\alpha)$, obtained from Eq. 2.105 with $\beta = 0$. In other words, the Q-function for ρ is obtained from the Wigner function of the same state after adding vacuum noise. As already illustrated in Fig. 2.10, heterodyne detection with an IQ mixer adds (ideally) vacuum noise to the signal before detection. This leads to the conclusion that the probability distributions for the simultaneous measurement of two orthogonal quadratures in heterodyne detection is given by the marginals of the Husimi-Q distribution rather than the of the Wigner function.

2.2.6.3 Dispersive qubit readout

1. Steady-state intra-cavity field

As discussed in light matter interaction in circuit QED section, in the dispersive regime the transmon-resonator Hamiltonian is well approximated by

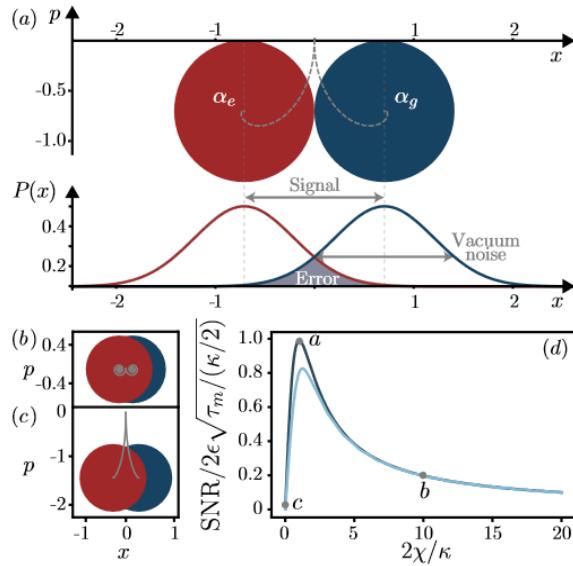
$$\hat{H}_{\text{disp}} \approx \hbar (\omega_r + \chi \hat{\sigma}_z) \hat{a}^\dagger \hat{a} + \frac{\hbar \omega_q}{2} \hat{\sigma}_z \quad (2.109)$$

To simplify the discussion, here we have truncated the transmon Hamiltonian to its first two levels, absorbed Lamb shifts in the system frequencies, and neglected a transmon-induced nonlinearity of the cavity (K_a). As made clear by the first term of the above expression, in

the dispersive regime, the resonator frequency becomes qubit-state dependent: If the qubit is in $|g\rangle$ then $\langle \hat{\sigma}_z \rangle = -1$ and the resonator frequency is $\omega_r - \chi$. On the other hand, if the qubit is in $|e\rangle$, $\langle \hat{\sigma}_z \rangle = 1$ and ω_r is pulled to $\omega_r + \chi$. In this situation, driving the cavity results in a qubit-state dependent coherent state, $|\alpha_{g,e}\rangle$. Thus, if the qubit is initialized in the superposition $c_g|g\rangle + c_e|e\rangle$, the system evolves to an entangled qubit-resonator state of the form

$$c_g |g, \alpha_g\rangle + c_e |e, \alpha_e\rangle \quad (2.110)$$

To interpret this expression, let us recall the paradigm of the Stern-Gerlach experiment. There, an atom passes through a magnet and the field gradient applies a spin-dependent force to the atom that entangles the spin state of the atom with the momentum state of the atom (which in turn determines where the atom lands on the detector). The experiment is usually described as measuring the spin of the atom, but in fact it only measures the final position of that atom. However, since the spin and position are entangled, we can uniquely infer the spin from the position, provided there is no overlap in the final position distributions for the two spin states. In this case we have effectively performed a projective measurement of the spin.



By analogy, if the spin-dependent coherent states of the microwave field, $\alpha_{e,g}$, can be resolved by heterodyne detection, then they act as pointer states in the qubit measurement. Moreover, since \hat{H}_{disp} commutes with the observable that is measured, $\hat{\alpha}_z$, this is QND (quantum non-demolition) measurement (in contrast to the Stern-Gerlach measurement which is destructive). Note that for a system initially in a superposition of eigenstates of the measurement operator, a QND measurement *does* in fact change the state by randomly collapsing

it onto one of the measurement eigenstates. The true test of 'QNDness' is that subsequent measurement results are not random but simply reproduce the first measurement result.

The objective in a qubit measurement is to maximize the readout fidelity in the shortest possible measurement time. To see how this goal can be reached, it is useful to first evaluate more precisely the evolution of the intra-cavity field under such a measurement. The intra-cavity field is obtained from Langevin equation Eq. 2.65 with $\hat{H}_S = \hat{H}_{disp}$ and by taking into account the cavity drive as discussed in section of controlling quantum systems with microwave drives. Doing so, we find that the complex field amplitude given that $\langle \hat{a} \rangle_\sigma = \alpha_\sigma$ the qubit is in state $\sigma = \{g, e\}$ satisfies

$$\begin{aligned}\dot{\alpha}_e(t) &= -i\varepsilon(t) - (\delta_r + \chi)\alpha_e(t) - \kappa\alpha_e(t)/2, \\ \dot{\alpha}_g(t) &= -i\varepsilon(t) - (\delta_r - \chi)\alpha_g(t) - \kappa\alpha_g(t)/2,\end{aligned}\quad (2.111)$$

with $\delta_r = \omega_r - \omega_d$ the detuning of the measurement drive to the bare cavity frequency. The time evolution of these two cavity fields in phase space are illustrated for three different values of $2\chi/\kappa$ by dashed gray lines in Fig. 2.12.

Focusing for the moment on the steady-state ($\dot{\alpha}_\sigma = 0$) response

$$\alpha_{e/g}^s = \frac{-\varepsilon}{(\delta_r \pm \chi) - i\kappa/2} \quad (2.112)$$

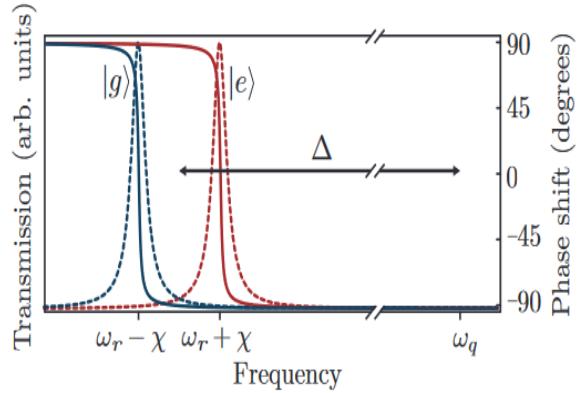
with $+$ for e and $-$ for g , results in the steady-state intra-cavity quadratures

$$A_{e/g}^s = \sqrt{\langle \hat{X} \rangle_{e/g}^2 + \langle \hat{P} \rangle_{e/g}^2} = \frac{2\varepsilon}{\sqrt{(\kappa/2)^2 + (\delta_r \pm \chi)^2}} \quad (2.113)$$

and phase

$$\phi_{e/g}^s = \arctan \left(\frac{\langle \hat{X} \rangle_{e/g}}{\langle \hat{P} \rangle_{e/g}} \right) = \arctan \left(\frac{\delta_r + \chi}{\kappa/2} \right) \quad (2.114)$$

These two quantities are plotted in Fig. 2.13. As could already have been expected from the form of \hat{H}_{disp} , a coherent tone of frequency $\omega_r \pm \chi$ on the resonator is largely transmitted if the qubit is in the ground (excited) state, and mostly reflected if the qubit is in the excited (ground) state. Alternatively, driving the resonator at its bare frequency ω_r leads to a different phase accumulation for the transmitted signal depending on the state of the qubit. In particular, on resonance with the bare resonator, $\delta_r = 0$, the phase shift of the signal associated to the two qubit states is simply $\pm \arctan(2\chi/\kappa)$. As a result, in the dispersive regime, measuring the amplitude and/or the phase of the transmitted or reflected signal from the resonator reveals information about the qubit state. On the other hand, when driving the resonator at the qubit frequency, for example, to realize a logical gate discussed further, the phase shift of the resonator field only negligibly depends on the qubit state. This results in negligible entanglement between the resonator, and consequently on negligible measurement-induced dephasing on the qubit.



It is very important to note that for purposes of simplification, all of the above discussion has been couched in terms of the amplitude and phase of the oscillating electric field internal to the microwave resonator. In practice, we can typically only measure the field externally in the transmission line(s) coupled to the resonator. The relation between the two is the subject of input-output theory. The main ideas can be summarized rather simply. Consider an asymmetric cavity with one port strongly coupled to the environment and one port weakly coupled. If driven from state of the qubit is in the field radiated by the cavity into the strongly coupled port. The same is true if the cavity is driven from the strongly coupled side, but now the output field is a superposition of the directly reflected drive plus the field radiated by the cavity. If the drive frequency is swept across the cavity resonance, the signal undergoes a phase shift of π in the former case and 2π in the latter. This affects the sensitivity of the output field to the dispersive shift induced by the qubit. If the cavity is symmetric, then half the information about the state appears at each output port so this configuration is inefficient.

2.2.6.4 Signal-to-noise ratio and measurement fidelity

Except for the last paragraph, the above discussion concerned the steady-state intra-cavity field from which we can infer the steady-state heterodyne signal. It is, however, also crucial to consider the temporal response of the resonator's output field to the measurement drive since, in the context of the quantum computing, qubit readout should be as fast as possible. Moreover, the probability of assigning the correct outcome to a qubit measurement, or more simply put the measurement fidelity, must also be large. As following discussion hopes to illustrate, simultaneously optimizing these two important quantities requires some care.

As discussed in microwave field detection section, the quadratures $\hat{X}_f(t)$ and $\hat{P}_f(t)$ are extracted from heterodyne measurement of the resonator output field. Combining these signals and integrating for a time τ_m , the operator corresponding to this measurement takes the form

$$\begin{aligned} \hat{M}(\tau_m) = & \int_0^{\tau_m} dt \left\{ w_X(t) \left[V_{IF} \hat{X}_f(t) + \hat{V}_{\text{noise}, X_f}(t) \right] \right. \\ & \left. + w_P(t) \left[V_{IF} \hat{P}_f(t) + \hat{V}_{\text{noise}, P_f}(t) \right] \right\} \end{aligned} \quad (2.115)$$

where $\hat{V}_{noise,X_f/P_f}(t)$ is the noise in the X_f/P_f quadrature. The weighting functions $\omega_X(t) = |\langle \hat{X}_f \rangle_e - \langle \hat{X}_f \rangle_g|$ and $\omega_P(t) = |\langle \hat{P}_f \rangle_e - \langle \hat{P}_f \rangle_g|$ are chosen such as to increase the discrimination of the two qubit states. Quite intuitively, because of qubit relaxation, these functions give less weight to the cavity response at long times since it will always reveal the qubit to be in its ground state irrespective of the prepared state. Moreover, for the situation illustrated in Fig. 2.12, there is no information on the qubit state in the P quadrature. Reflecting this, $\omega_P(t) = 0$ which prevents the noise in that quadrature from being integrated. Following microwave field detection and phase-space representations and their relation to field detection sections, the probability distribution for the outcome of multiple shots of the measurement of $\hat{M}(\tau_m)$ is expected to be Gaussian and characterized by the marginal of the Q-function of the intra-cavity field. Using the above expression, the signal-to-noise ratio (SNR) of this measurement can be defined as illustrated in Fig. 2.12 (a) for the intra-cavity field: it is the separation of the average combined heterodyne signals corresponding to the two qubit states divided by the standard deviation of the signal, an expression which takes the form

$$\text{SNR}^2(t) \equiv \frac{\left| \langle \hat{M}(t) \rangle_e - \langle \hat{M}(t) \rangle_g \right|^2}{\left\langle \hat{M}_N^2(t) \right\rangle_e + \left\langle \hat{M}_N^2(t) \right\rangle_g} \quad (2.116)$$

Here, $\langle \hat{M} \rangle_\sigma$ is the average integrated heterodyne signal given that the qubit is in state σ , and $M_N = M - \langle M \rangle$ the noise operator which takes into account the added noise but also the intrinsic vacuum noise of the quantum states. In addition to the SNR, another important quantity is the measurement fidelity.

$$F_m = 1 - [P(e | g) + P(g | e)] \equiv 1 - E_m \quad (2.117)$$

where $P(\sigma | \sigma')$ is the probability that a qubit in state σ is measured to be in state σ' . In the second equality, we have defined the measurement error E_m which, as illustrated in Fig. 2.12 (a), is simply the overlap of the marginals $P_\sigma(x)$ of the Q-functions for the two qubit states. This can be expressed as $E_m = \int dx_{\phi_{\text{LO}}+\pi/2} \min[P_0(x_{\phi_{\text{LO}}+\pi/2}), P_1(x_{\phi_{\text{LO}}+\pi/2})]$ where the LO phase is chosen to minimize E_m . Using the expression, the measurement fidelity is found to be related to SNR by $F_m = 1 - \text{erfc}(\text{SNR}/2)$, where erfc is that complementary error function. It is important to note that this last result is valid only if the marginals are Gaussian. In practice, qubit relaxation and higher-order effects omitted in the dispersive Hamiltonian Eq. 2.109 can lead to distortion of the coherent states and therefore to non-Gaussian marginals. Kerr-type nonlinearities that are common in circuit QED tend to create a banana-shaped distortion of the coherent states in phase space, something that is sometimes referred to bananization. Although we are interested in short measurement times, it is useful to consider the simpler expression for the longtime behaviour of the SNR which suggests different strategies to maximize the measurement fidelity. Assuming $\delta_r = 0$ and ignoring the prefactors related to gain and mixing, we find

$$\text{SNR}(\tau_m \rightarrow \infty) \simeq (2\varepsilon/\kappa)\sqrt{2\kappa\tau_m} |\sin 2\phi| \quad (2.118)$$

where ϕ is given by Eq. 2.114. The reader can easily verify that the choice $\chi/\kappa = 1/2$ maximizes Eq. 2.118. This ratio is consequently often chosen experimentally. While leading to a smaller steady-state SNR, other choices of the ratio χ/κ can be more advantageous at finite measurement times. In the small χ limit, the factor $2\epsilon/\kappa$ in $\text{SNR}(\tau_m \rightarrow \infty)$ can be interpreted using Eq. 2.111 as the square-root of the steady-state average intra-cavity measurement photon number. Another approach to improve the SNR is therefore to work at large measurement photon number \bar{n} . This idea, however, cannot be pushed too far since increasing the measurement photon number leads to a breakdown of the approximations that have been used to derive the dispersive Hamiltonian Eq. 2.109. Indeed, as discussed in dispersive regime section, the small parameter in the perturbation theory that leads to the dispersive approximation is not g/Δ but rather \bar{n}/n_{crit} , with n_{crit} the critical photon number introduced in Eq. 2.33. Well before reaching $\bar{n}/n_{crit} \sim 1$, higher-order terms in the dispersive approximation start to play a role and lead to departures from the expected behavior. For example, it is commonly experimentally observed that the dispersive measurement loses its QND character well before $\hat{n} \sim n_{crit}$ and often at measurement photon populations as small as $\hat{n} \sim 1 - 10$. Because of these spurious qubit flips, measurement photon numbers are typically chosen to be well below n_{crit} . While this non-QNDness at $\hat{n} < n_{crit}$ is expected from the discussion of dressed-dephasing, the predicted measurement-induced qubit flip rates are smaller than often experimentally observed. We note that qubit transitions at $\hat{n} > n_{crit}$ caused by accidental resonances within the qubit-resonator system. To reach high fidelities, it is also important for the measurement to be fast compared to the qubit relaxation time T_1 . A strategy to speed-up the measurement is to use a low-Q oscillator which leads to a faster readout rate simply because the measurement photons leak out more rapidly from the resonator to be detected. However, this should not be done at the price of increasing the Purcell rate γ_κ to the point where this mechanism dominates qubit decay. It is possible to avoid this situation to a large extent by adding a Purcell filter at the output of the resonator. Fixing κ so as to avoid Purcell decay and working at the optimal χ/κ ratio, it can be shown that the steady-state response is reached in a time $\propto 1/\chi$. Large dispersive shifts can therefore help to speed-up the measurement. As can be seen Eq. 2.32, χ can be increased by working at larger qubit anharmonicity or, in other words, larger charging energy E_C . Once more, this cannot be pushed too far since the transmon charge dispersion and therefore its dephasing rate increase with E_C . The above discussion shows that QND qubit measurement in circuit QED is a highly constrained problem. The state-of-the-art for such measurements is currently of $F_m \sim 98.25\%$ in $\tau_m = 48$ ns, when minimizing readout time, and 99.2% in 88 ns, when maximizing the fidelity, in both cases using $\bar{n} \sim 2.5$ intra-cavity measurement photons. These results were obtained by detailed optimization of the system parameters following the concepts introduced above but also given an understanding of the full time response of the measurement signal $|\langle \hat{M}(t) \rangle_1 - \langle \hat{M}(t) \rangle_0|$. The main limitation in these reported fidelity was the relatively short qubit relaxation time of $7.6 \mu\text{s}$. Joint simultaneous dispersive readout of two transmon qubits capacitively coupled to the same resonator has also been realized. The very small photon number used in these experiments underscores the importance of quantum-limited amplifiers in the first stage of the measurement chain. Before the development of these amplifiers, which opened the possibility to perform strong single-shot (i.e. projective) measurements, the SNR in dispersive measurements was well below unity, forcing the results of these weak measurements to be averaged over tens of thousands of repetitions of the

experiment to extract information about the qubit state. The advent of near quantum-limited amplifiers has made it possible to resolve the qubit state in a single-shot something which has led, for example, to the observation of quantum jumps of a transmon qubit. Finally, we point out that the quantum efficiency, η , of the whole measurement chain can be extracted from the SNR using

$$\eta = \frac{SNR^2}{4\beta_m} \quad (2.119)$$

where $\beta_m = 2\chi \int_0^{\tau_m} dt \text{Im} [\alpha_g(t)\alpha_e(t)^*]$ is related to the measurement-induced dephasing discussed further. This connection between quantum efficiency, SNR, and measurement-induced dephasing results from the fundamental link between the rate at which information is gained in a quantum measurement and the unavoidable back action on the measured system.

3. Other approaches

a. Josephson Bifurcation Amplifier

While the vast majority of circuit QED experiments rely on the approach described above, several other qubit-readout methods have been theoretically explored or experimentally implemented. One such alternative is known as the Josephson Bifurcation Amplifier (JBA) and relies on using, for example, a transmission-line resonator that is made nonlinear by incorporating a Josephson junction in its center conductor. This circuit can be seen as a distributed version of the transmon qubit and is well described by the Kerr-nonlinear Hamiltonian of Eq. 2.13. With a relatively weak Kerr nonlinearity (~ -500 kHz) and under a coherent drive of well chosen amplitude and frequency, this system bifurcates from a low photon-number state high photon-number state. By dispersively coupling a qubit to the nonlinear resonator, this bifurcation can be made qubit-state dependent. It is possible to exploit the fact that the low-and high-photon-number states can be easily distinguished to realize high-fidelity single-shot qubit readout.

b. High-power readout and qubit 'punch out'

Coming back to linear resonators, while the non-QNDness at moderate measurements photon number mentioned above leads to small measurement fidelity, it was observed by a fearless graduate student that, in the limit of very large measurement power, a fast and high-fidelity single-shot readout is recovered. An intuitive understanding of this observation can be obtained from the Jaynes-Cummings Hamiltonian Eq. 2.22. Indeed, for $n \gg \sqrt{n}$, the first term of this Hamiltonian dominates over the qubit-oscillator interaction $\propto g$ such that the cavity responds at its bare frequency ω_r despite the presence of the transmon. This is sometimes referred to as '*punching out*' the qubit and can be understood as a quantum-to-classical transition where, in the correspondence limit, the system behaves classically and therefore responds at which this transition occurs depends on the state of the transmon, leading to a high-fidelity measurement. This high-power readout is, however,

obtained at the expense of completely losing the QND nature of the dispersive readout.

c. *Squeezing*

Finally, the \sqrt{n} scaling of SNR ($\tau_m \rightarrow \infty$) mentioned above can be interpreted as resulting from populating the cavity with a coherent state and is known as the standard quantum limit. It is natural to ask if replacing the coherent measurement tone with squeezed input radiation can lead to Heisenberg-limited scaling for which the SNR scales linearly with the measurement photons number. To achieve this, one might imagine squeezing a quadrature of the field to reduce the overlap between the two pointer states. In Fig. 2.12, this corresponds to squeezing along X . The situation is not so simple since the large dispersive coupling required for high-fidelity qubit readout leads to a significant rotation of the squeezing angle as the pointer states evolve from the center of phase space to their steady-state. This rotation results in increased measurement noise due to contributions from the anti-squeezed quadrature. Borrowing the idea of quantum-mechanics-free subsystems, it has been shown that Heisenberg-limited scaling can be reached with two-mode squeezing by dispersively coupling the qubit to two rather than one resonator.

d. *Longitudinal readout*

An alternative approach to qubit readout is based on the Hamiltonian \hat{H}_z of Eq. 2.52 with its longitudinal qubit-oscillator coupling $g_z(\hat{a}^\dagger + \hat{a})\hat{\sigma}_z$. In contrast to the dispersive Hamiltonian which leads to a rotation in phase space, longitudinal coupling generates a linear displacement of the resonator field that is conditional on the qubit state. As a result, while under the dispersive evolution there is little information gain about the qubit state at short times (see the poor pointer state separation at short times in Fig. 2.12(a)), \hat{H}_z rather generates the ideal dynamics for a measurement with a 180° out-of-phase displacements of the pointer states α_g and α_e . It is therefore expected that this approach can lead to much shorter measurement times than is possible with the dispersive readout. Another advantage is that \hat{H}_z commutes with the measured observable, $[\hat{H}_z, \hat{\sigma}_z] = 0$, corresponding to a QND measurement. While the dispersive Hamiltonian \hat{H}_{disp} also commutes with $\hat{\alpha}_z$, it is not the case for the full Hamiltonian Eq. 2.20 from which \hat{H}_{disp} is perturbatively derived. As already discussed, this non-QNDness leads to Purcell decay and to a breakdown of the dispersive approximation when the photon populations is not significantly smaller than the critical photon number n_{crit} . On the other hand, because \hat{H}_z is genuinely QND it does not suffer from these problems and the measurement photon number can, in principle, be made larger under longitudinal than under dispersive coupling. Moreover, given that \hat{H}_z leads to displacement of the pointer states rather than to rotation in phase space, single-mode squeezing can also be used to increase the measurement SNR. Because the longitudinal coupling can be thought of as a cavity drive of amplitude $\pm g_z$ with the sign being conditional on the qubit state, \hat{H}_z leads in steady-state to a pointer state displacement $\pm g_z/(\omega_r + i\kappa/2)$, see Eq. 2.112. With $\omega_r \gg g_z$, κ in practice, this displacement is negligible and cannot realistically be used for qubit readout. One approach to increase the pointer state separation is to activate the longitudinal coupling by modulating g_z at the resonator frequency. Taking $g_z(t) = \tilde{g}_z \cos(\omega_r t)$

leads, in a rotating frame and after dropping rapidly oscillating terms, to the Hamiltonian

$$\tilde{H}_z = \frac{\tilde{g}_z}{2}(\hat{a}^\dagger + a)\hat{\sigma}_z \quad (2.120)$$

Under this modulation, the steady-state displacement now becomes $\pm\tilde{g}_z/\kappa$ and can be significant even for moderate modulation amplitudes \tilde{g}_z . Circuits realizing the longitudinal coupling with transmon or flux qubits have been studied. Another approach to realize these ideas is to strongly drive a resonator dispersively coupled to a qubit. Indeed, the strong drive leads to large displacement of the cavity field $\hat{a} \rightarrow \hat{a} + \alpha$ which on the dispersive Hamiltonian leads to

$$\chi\hat{a}^\dagger\hat{a}\hat{\sigma}_z \rightarrow \chi\hat{a}^\dagger\hat{a}\hat{\sigma}_z + \alpha\chi(\hat{a}^\dagger + \hat{a})\hat{\sigma}_z + \chi\alpha^2\hat{\sigma}_z \quad (2.121)$$

where we have assumed α to be real for simplicity. For χ small and α large, the second term longitudinal readout can be realized as a limit of the dispersive readout where χ approaches zero while α grows such that $\chi\alpha$ is constant. A simple interpretation of this observation is that, for strong drives, the circle on which the pointer states rotate due to the dispersive interaction has a very large radius α such that, for all practical purposes, the motion appears linear. A variation of this approach which allows for larger longitudinal coupling strength was experimentally realized and relies on driving the qubit at the frequency of the resonator. This is akin to the cross-resonance gate discussed further and which leads to the desired longitudinal interaction. A more subtle approach to realize a synthetic longitudinal interaction is to drive a qubit with a Rabi frequency Ω_R while driving the resonator at the sideband frequencies $\omega_r \pm \Omega_R$. This idea was implemented and showed improvement of qubit readout with single-mode squeezing. Importantly, because these realizations are based on the dispersive Hamiltonian, they suffer from Purcell decay and non-QNDness. Circuits realizing dispersive-like interactions that are not derived from a Jaynes-Cummings interaction have been studied.

2.2.7 Qubit-Resonator Coupling Regimes

We now turn to a discussion of the different coupling regimes that are accessible in circuit QED and how these regimes are probed experimentally. We first consider the resonant regime where the qubit is tuned in resonance with the resonator, before moving on to the dispersive regime characterized by large qubit-resonator detuning. While the situation of most experimental interest is the strong coupling regime where the coupling strength g overwhelms the decay rates, we also touch upon the so-called bad-cavity and bad-qubit limits because of their historical importance and their current relevance to hybrid quantum systems. Finally, we briefly consider the ultra strong coupling regime where g becomes comparable or is even larger than the system's frequencies. To simplify the discussion, we will treat the artificial atom as a simple two-level system throughout this section.

2.2.7.1 Resonant regime

The low-energy physics of the Jaynes-Cummings model is well described by the ground state $|g, 0\rangle = |g, 1\rangle$ and first two excited states

$$\begin{aligned} |\overline{g, 1}\rangle &= (|g, 1\rangle - |e, 0\rangle)/\sqrt{2} \\ |e, 0\rangle &= (|g, 1\rangle + |e, 0\rangle)/\sqrt{2} \end{aligned} \quad (2.122)$$

which are split in frequency by $2g$. As discussed in the context of the dispersive readout, the coupled qubit-resonator system can be probed by applying a coherent microwave tone to the input of the resonator and measuring the transmitted or reflected signal. To arrive at an expression for the expected signal in such an experiment, we consider the equations of motion for the field and qubit annihilation operators in the presence of a coherent drive of amplitude ϵ and frequency ω_d on the resonator's input port. In a frame rotating at the drive frequency, these equations take the form

$$\langle \hat{a} \rangle = -\left(\frac{\kappa}{2} + i\delta_r\right) \langle \hat{a} \rangle - ig \langle \hat{\sigma}_- \rangle - i\epsilon, \quad (2.123)$$

$$\dot{\langle \hat{\sigma}_- \rangle} = -(\gamma_2 + i\delta_q) \langle \hat{\sigma}_- \rangle + ig \langle \hat{a} \hat{\sigma}_z \rangle, \quad (2.124)$$

with $\delta_r = \omega_r - \omega_d$ and $\delta_q = \omega_q - \omega_d$, and where γ_2 is defined in Eq. 2.69. These expressions are obtained using $\partial_t \langle \hat{O} \rangle = Tr \dot{\rho} \hat{O}$ and the master equations at zero temperature and in the two-level approximation for the transmon. Alternatively, the expression for $\partial_t \langle \hat{a} \rangle$ is simply the average of Eq. ?? with \hat{H}_S the Jayne-Cummings Hamiltonian.

At very low excitation amplitude ϵ , it is reasonable to truncate the Hilbert space to the first three levels defined above. In this subspace, $\langle \hat{a} \hat{\sigma}_z \rangle = -\langle \hat{a} \rangle$ since \hat{a} acts nontrivially only in the qubit is in the ground state. It is then simple to compute the steady-state transmitted homodyne power by solving the above expressions with $\partial_t \langle \hat{a} \rangle = \partial_t \langle \hat{\sigma}_- \rangle = 0$ and using Eq. 2.113 to find

$$|A|^2 = \left(\frac{\epsilon V_{IF}}{2}\right)^2 \left| \frac{\delta_q - i\gamma_2}{(\delta_q - i\gamma_2)(\delta_r - i\kappa/2) - g^2} \right|^2 \quad (2.125)$$

This expression is exact in the low excitation power limit. Taking the qubit and the oscillator to be on resonance $\Delta = \omega_q - \omega_r = 0$, we now consider the result of cavity transmission measurements in the three different regimes of qubit-cavity interaction.

1. Bad-cavity limit

We first consider the bad-cavity limit realized when the cavity decay rate overwhelms the coupling g which is itself larger than the qubit linewidth: $\kappa > g \gg \gamma_2$. This situation corresponds to an overdamped oscillator and, at qubit-oscillator resonance, leads to rapid decay of the qubit. A simple model for this process is obtained using the truncated Hilbert space discussed above where we now drop the cavity drive for simplicity. Because of the very

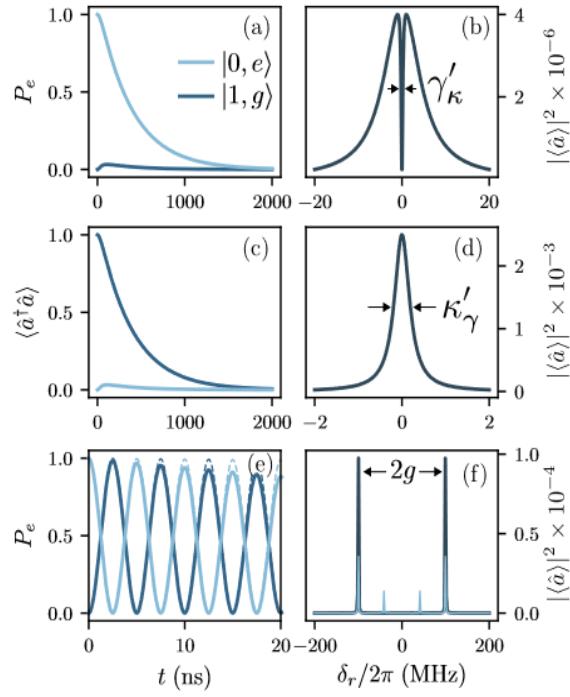
large decay rate κ , we can assume the oscillator to rapidly reach its steady-state $\partial_t \langle \hat{a} \rangle = 0$. Using the resulting expression for $\langle \hat{a} \rangle$ in Eq. 2.124 immediately leads to

$$\langle \dot{\hat{\sigma}}_- \rangle = - \left(\frac{\gamma_1 + \gamma'_k}{2} + \gamma_\varphi \right) \langle \hat{\sigma}_- \rangle \quad (2.126)$$

where we have defined the Purcell decay rate $\gamma'_\kappa = 4g^2/\kappa$. The expression for this rate has a rather different form than the Purcell rate $\gamma_\kappa = (g/\Delta)^2\kappa$. These two results are, however, not incompatible but have been obtained in very different regimes. An expression for the Purcell rate that interpolates between the two above expressions can be obtained and takes the form $\kappa g^2/[(\kappa/2)^2 + \Delta^2]$. The situation described here is illustrated for $\kappa/g = 10$ and $\gamma_1 = 0$ in Fig. 2.14(b) shows the transmitted power versus drive frequency in the presence of a very weak coherent tone populating the cavity with $\hat{n} \ll 1$ photons. The response shows a broad Lorentzian peak of width κ together with a narrow electromagnetically induced transparency (EIT)-like window of width γ'_κ . This effect which is due to interference between the intra-cavity field and the probe tone vanishes in the presence of qubit dephasing. Although not the main regime of interest in circuit QED, the bad-cavity limit offers an opportunity to engineer the dissipation seen by the qubit. For example, this regime has been used to control the lifetime of long-lived donor spins in silicon in a hybrid quantum system.

2.Bad-qubit limit

The bad qubit limit corresponds to the situation where a high-Q cavity with large qubit-oscillator coupling is realized, while the qubit dephasing and/or energy relaxation rates is large: $\gamma_2 > g \gg \kappa$. Although this situation is not typical of circuit QED with transmon qubits, it is relevant for some hybrid systems that suffer from significant dephasing. This is the case, for example, in early experiments with charge qubits based on semiconductor quantum dots coupled to superconducting resonators. In analogy to the bad-cavity case, the strong damping of the qubit together with the qubit-resonator coupling leads to the photon decay rate $\kappa'_\gamma = 4g^2/\gamma_1$ which is sometimes known as the 'inverse' Purcell rate. In this situation, the cavity response is a simple Lorentzian broadened by the inverse Purcell rate. If the qubit were to be probed directly rather than indirectly via the cavity, the atomic response would show the EIT-like feature of Fig. 2.14(b), now with a dip of width κ'_γ . The reader should also be aware that qubit-resonator detuning-dependent dispersive shifts of the cavity resonance can be observed in this bad-qubit limit. The observation of such dispersive shifts on its own should not be mistaken for an observation of strong coupling.

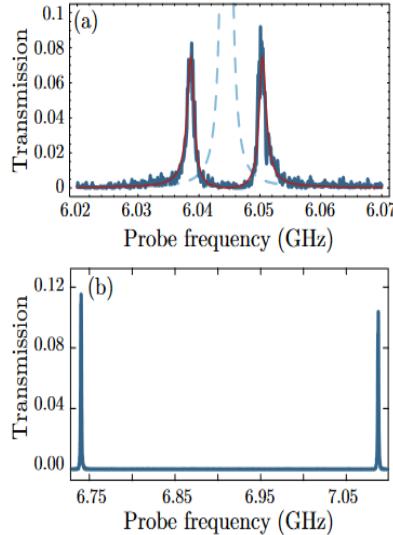


3. Strong coupling regime

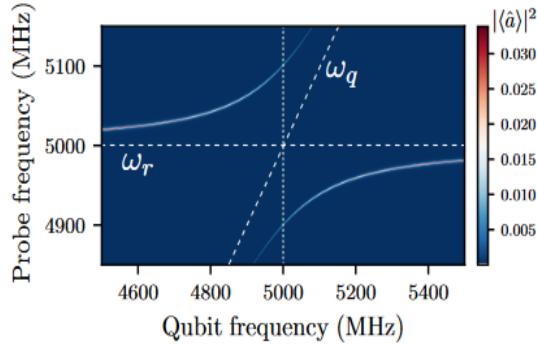
We now turn to the case where the coupling strength overwhelms the qubit and cavity decay rates, $g > \kappa, \gamma_2$. In this regime, light-matter interaction is strong enough for a single quantum to be coherently exchanged between the electromagnetic field and the qubit before it is irreversibly lost to the environment. In other words, at resonance $\Delta = 0$ the splitting $2g$ between the two dressed eigenstates $|\overline{g}, \overline{1}\rangle, |\overline{e}, \overline{0}\rangle$ of Eq. 2.122 is larger than their linewidth $\kappa/2 + \gamma_2$ and can be resolved spectroscopically. We note that, with the eigenstates being half-photon and half-qubit, the above expression for the dressed-state linewidth is simply the average of the cavity and of the qubit linewidth. Figure 2.14(f) shows cavity transmission for $(\kappa, \gamma_1, \gamma_\varphi)/g = (0.1, 0.1, 0)$ and at low excitation power such that, on average, there is significantly less than one photon in the cavity. The resulting doublet of peaks located at $\omega_r \pm g$ is the direct signature of the dressed-states $|\overline{g}, \overline{1}\rangle, |\overline{e}, \overline{0}\rangle$ and is known as the vacuum Rabi splitting. The observation of this doublet is the hallmark of the strong coupling regime. The first observation of this feature in cavity QED with a single atom and a single photon was reported by Thompson in 1992. In this experiment, the number of atoms in the cavity was not well controlled and it could only be determined that there was *on average* one atom in interaction with the cavity field. This distinction is important because, in the presence of N atoms, the collective interaction strength is $g\sqrt{N}$ and the observed splitting correspondingly larger. Atom number fluctuation is obviously not a problem in circuit QED and, with the very strong coupling regime is not particularly challenging in this system. In fact, the very first circuit QED experiment of Wallraff in 2004 reported the observation of a clear vacuum Rabi splitting with $2g/(\kappa/2 + \gamma_2) \sim 10$. This first demonstration used a

charge qubit which, by construction, has a much smaller coupling g than typical transmon qubits. As a result, more recent experiments with transmon qubits can display ratios of peak separation to linewidth in the several hundred.

Figure 2.16 shows the qubit-oscillator spectrum as a function of probe frequency, as above, but now also as a function of the qubit frequency allowing to see the full qubit-resonator avoided crossing. The horizontal dashed line corresponds to the bare cavity frequency while the diagonal dashed line is the bare qubit frequency. The vacuum Rabi splitting of Fig. 2.14(f) is obtained from a line cut (dotted vertical line) at resonance between ω_r . Because it is the cavity that is probed here, the response is larger when the dressed-states are mostly cavity-like and disappears away from the cavity frequency where the cavity longer responds to the probe.

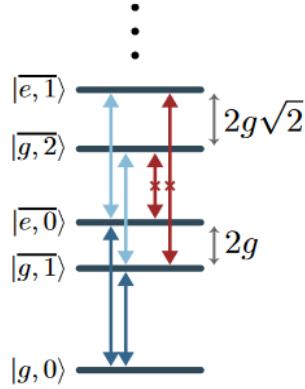


It is interesting to note that splitting predicted by Eq. 2.125 for the transmitted homodyne signal is in fact smaller than $2g$ in the presence of finite relaxation and dephasing. Although not significant in circuit QED with transmon qubits, this correction can become important in systems such as charge qubits in quantum dots that are not very deep in the strong coupling regime. We also note that the observed splitting can be smaller when measured in reflection rather than in transmission.



Rather than spectroscopic measurements, strong light-matter coupling can also be displayed in time-resolved measurements. Starting from the qubit-oscillator ground state, this can be done, for example, by first pulsing the qubit to first excited state and then bringing it on resonance with the cavity. As illustrated in Fig. 2.14(e), this results in oscillations in the qubit and cavity populations at the vacuum Rabi frequency $2g$. Time-resolved vacuum Rabi oscillations in circuit QED were first performed with a flux qubit coupled to a discrete LC oscillator realized in the bias circuitry of the device. This experiment was followed by a similar observation with a phase qubit coupled to a coplanar waveguide resonator. In the limit of weak excitation power which we have considered so far, the coupled qubit-oscillator system is indistinguishable from two coupled classical linear oscillators. As a result, while the dressed-states that are probed in these experiments are entangled, the observation of an avoided crossing cannot be taken as a conclusive demonstration that the oscillator field is quantized or of qubit-oscillator entanglement. Indeed, a vacuum Rabi splitting can be interpreted as the familiar normal mode splitting of two coupled classical oscillators. A clear signature of the quantum nature of the system can, however, be obtained by probing the \sqrt{n} dependence of the spacing of the higher excited states of the Jaynes-Cummings ladder already discussed in the Jaynes-Cummings spectrum section. This dependence results from the matrix element of the operator \hat{a} and is consequently linked to the quantum nature of the field. Experimentally, these transitions can be accessed in several ways including by two-tone spectroscopy, by increasing the probe tone power, or by increasing the system temperature. The light blue line in Fig. 2.14(f) shows cavity transmission with a thermal photon number of $\bar{\eta}_\kappa = 0.35$ rather than $\bar{\eta}_\kappa = 0$ (dark blue line). At this more elevated temperature, additional pairs of peaks with smaller separation are now observed in addition to the original peaks separated by $2g$. As illustrated in Figure 2.17, these additional structures are due to multi-photon transitions and their \sqrt{n} scaling reveal the anharmonicity of the Jaynes-Cummings ladder. Interestingly, the matrix elements of the transitions that lie outside original vacuum Rabi splitting peaks are suppressed and these transitions are therefore not observed, see the red arrow in Fig. 2.17. We also note that, at much larger power or at elevated temperature, the system undergoes a quantum-to-classical transition and a single peak at the resonator frequency ω_r is observed. In short the impact of the qubit on the system is washed away in the correspondence limit. This is to be expected from the form of the Jaynes-Cummings Hamiltonian Eq. 2.22 where the qubit-cavity coupling $\hbar g(\hat{a}^\dagger \sigma_- + \hat{a} \sigma_+)$ with its \sqrt{n} scaling is overwhelmed by the free cavity Hamiltonian $\hbar \omega_r \hat{a}^\dagger \hat{a}$ which scales as n . This is the same

mechanism that leads to the high-power readout.



Beyond this spectroscopic evidence, the quantum nature of the field and qubit-oscillator entanglement was also demonstrated in a number of experiments directly measuring the joint density matrix of the dressed states. For example, this is achieved by creating one of the entangled states $\{|g, 1\rangle, |e, 0\rangle\}$ in a time-resolved vacuum Rabi oscillation experiment and, subsequently, measuring the qubit state in a dispersive measurement and the photon state using a linear detection method. A range of experiments used the ability to create entanglement between a qubit and a photon through the resonant interaction with a resonator, e.g. in the context of quantum computation, for entangling resonator modes, and transferring quantum states.

2.2.7.2 Dispersive regime

For most quantum computing experiments, it is common to work in the dispersive regime where as already discussed in earlier section, the qubit is strongly detuned from the oscillator with $|\Delta| \gg g$. There, the dressed eigenstates are only weakly entangled qubit-oscillator states. This is to be contrasted to the resonant regime where these eigenstates are highly entangled resulting in the qubit and the oscillator to completely lose their individual character. In the two-level system approximation, the dispersive regime is well described by the Hamiltonian \hat{H}_{disp} . There, we had interpreted the dispersive coupling as a qubit-state dependent shift of the oscillator frequency. This shift can be clearly seen in Fig.2.16 as the deviation of the oscillator response from the bare oscillator frequency away from resonance (horizontal dashed line). This figure also makes it clear that the qubit frequency, whose bare value is given by the diagonal dashed line, is also modified by the dispersive coupling to the oscillator. To better understand this qubit-frequency shift, it is instructive to rewrite \hat{H}_{disp} as

$$\hat{H}_{\text{disp}} \approx \hbar\omega_r \hat{a}^\dagger \hat{a} + \frac{\hbar}{2} \left[\omega_q + 2\chi \left(\hat{a}^\dagger \hat{a} + \frac{1}{2} \right) \right] \hat{\sigma}_z \quad (2.127)$$

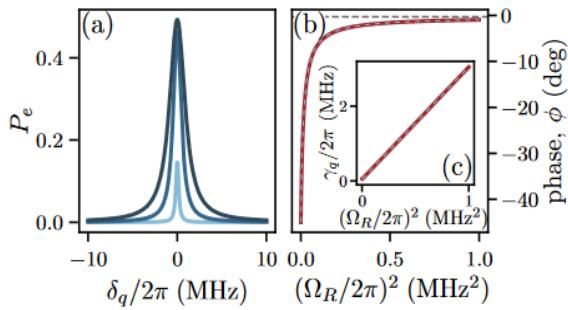
where it is now clear that the dispersive interaction of amplitude χ not only leads to a qubit-state dependent frequency pull of the oscillator, but also to a photon number dependent frequency shift of the qubit given by $2\chi \hat{a}^\dagger \hat{a}$. This is known as the ac-Stark shift (or

the quantized light shift) and is here accompanied by a Lamb shift corresponding to the factor of $1/2$ in the last term of Eq. 2.127 and which we had dropped in Eq. 2.109. In this section, we explore some consequences of this new point of view on the dispersive interaction, starting by the first reviewing some of the basic aspects of qubit spectroscopic measurements.

1.Qubit Spectroscopy

To simplify the discussion, we first consider spectroscopically probing the qubit assuming that the oscillator remains in its vacuum state. This is done by applying a coherent field of amplitude α_d and frequency ω_d to the qubit, either via a dedicated voltage gate on the qubit or to the input port of the resonator. Ignoring the resonator. Ignoring the resonator for the moment, this situation is described by the Hamiltonian $\delta_q \hat{\sigma}_z/2 + \Omega_R \hat{\sigma}_x/2$, where $\delta_q = (\omega_q + \chi) - \omega_d$ is the detuning between the Lamb-shifted qubit transition frequency and the drive frequency, and $\Omega_R \propto \alpha_d$ is the Rabi frequency. Under this Hamiltonian and using the master equation Eq. 2.68 projected on two levels of the qubit, the steady-state probability $P_e = (\langle \hat{\sigma}_z \rangle_s + 1)/2$ for the qubit to be in its excited state (or, equivalently, the probability to be in the ground state, P_g) is found to be

$$P_e = 1 - P_g = \frac{1}{2} \frac{\Omega_R^2}{\gamma_1 \gamma_2 + \delta_q^2 \gamma_1 / \gamma_2 + \Omega_R^2} \quad (2.128)$$

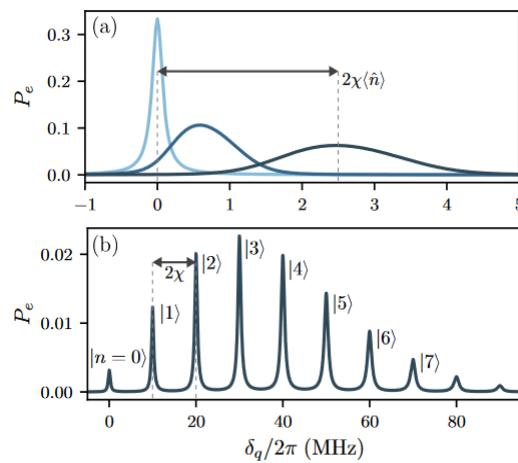


The Lorantzian lineshape of P_e as a function of the drive frequency is illustrated in Fig. 2.18(a). In the limit of strong qubit drive, i.e. large Rabi frequency Ω_R , the steady-state qubit population reaches saturation with $P_e = P_g = 1/2$, see Fig. 2.18(b). Moreover, as the power increases, the full width at half maximum (FWHM) of the qubit lineshape evolves from the bare qubit linewidth given by $\gamma_q = 2\gamma_2$ to $2\sqrt{1/T_2^2 + \Omega_R^2 T_1/T_2}$, something that is known as power broadening and which is illustrated in Fig. 2.18(c). In practice, the unbroadened dephasing rate γ_2 can be determined from spectroscopic measurements by extrapolating to zero spectroscopy tone power the linear dependence of ν_{HWHM}^2 . This quantity can also be determined in the time domain from a Ramsey fringe experiment. In typical optical spectroscopy of atoms in a gas, one directly measures the absorption of

photons by the gas as a function of the frequency of the photons. In circuit QED, one typically performs quantum jump spectroscopy by measuring the probability that an applied microwave drive places the qubit into its excited state. The variation in qubit population with qubit drive can be measured by monitoring the change in response of the cavity to the spectroscopy drive. As discussed in dispersive qubit readout section, this is realized using two-tone spectroscopy by measuring the cavity transmission, or reflection, of an additional drive of frequency close to ω_r . In the literature, this second drive is often referred to as the probe or measurement tone, while the spectroscopy drive is also known as the pump tone. As shown by Eq. 2.114, the phase of the transmitted probe tone is related to the qubit population. In particular, with the probe tone at the bare cavity frequency and in the weak dispersive limit $\chi \ll \kappa$, this phase is simply proportional to the qubit population, $\phi_s = \arctan(2\chi\langle\hat{\sigma}_z\rangle_s) \approx 2\chi\langle\hat{\sigma}_z\rangle_s/\kappa$. Monitoring ϕ_s as a function of the spectroscopy tone frequency therefore directly reveals the Lorentzian qubit lineshape.

2. AC-Stark shift and measurement-induced broadening

In the above discussion, we have implicitly assumed that the amplitude of the measure tone is such that the intra-cavity photon population is vanishingly small $\langle\hat{a}^\dagger\hat{a}\rangle \rightarrow 0$. As is made clear by Eq. 2.127, increase in photon population leads to a qubit frequency shift by an average value of $2\chi\langle\hat{a}^\dagger\hat{a}\rangle$. Figure 2.19(a) shows this ac-Stark shift in the steady-state qubit population as a function of spectroscopy frequency for three different probe drive powers. Taking advantage of the dependence of the qubit frequency on measurement power, prior knowledge of the value of χ allows one to infer the intra-cavity photon number as a function of input pump power from such measurements. However, care must be taken since the linear dependence of the qubit frequency on power predicted in Eq. 2.109 is only valid well inside the dispersive regime or, more precisely, at small \bar{n}/n_{crit} . We come back to this shortly.



As is apparent from Fig. 2.19(a), in addition to causing a frequency shift of the qubit, the cavity photon population also causes a broadening of the qubit linewidth. This can be

understood simply by considering again the form of \hat{H}_{disp} in Eq. 2.127. Indeed, while in the above discussion we considered only the *average* qubit frequency shift, $2\chi\langle\hat{a}^\dagger\hat{a}\rangle$, the actual shift is rather given by $2\chi\hat{a}^\dagger\hat{a}$ such that the full photon-number distribution is important. As a result, when the cavity is prepared in a coherent state by the measurement tone, each Fock state $|n\rangle$ of the coherent field leads to its own qubit frequency shift $2\chi n$. In the weak dispersive limit corresponding to χ/κ small, the observed qubit lineshape is thus the result of the inhomogeneous broadening due to the Poisson statistics of the coherent state populating the cavity. This effect becomes more apparent as the average measurement photon number \bar{n} increases and results in a crossover from a Lorentzian qubit lineshape whose linewidth scales with \bar{n} to a Gaussian lineshape whose linewidth rather scales as $\sqrt{\bar{n}}$. This square-root dependence can be traced to the coherent nature of the cavity field. For a thermal cavity field, a $\bar{n}(\bar{n}+1)$ dependence is rather expected and observed. This change in qubit linewidth due to photon shot noise in the coherent measurement tone populating the cavity can be interpreted as the unavoidable dephasing that a quantum system undergoes during measurement. Using a polaron-type transformation familiar from condensed-matter theory, the cavity can be integrated out of the qubit-cavity master equation and, in this way, the associated measurement-induced dephasing rate can be expressed in the dispersive regime as $\gamma_m(t) = 2\chi \text{Im} [\alpha_g(t)\alpha_e^*(t)]$, where $\alpha_{g/e}(t)$ are the two qubit states. In the long time limit, the above rate can be expressed in the more intuitive form $\gamma_m = \kappa|\alpha_e - \alpha_g|^2/2$, where $\alpha_e - \alpha_g$ is the distance between the two steady-state pointer states. Unsurprisingly, measurement-induced dephasing is faster when the pointer states are more easily distinguishable and the measurement thus more efficient. This last expression can also be directly obtained from the entangled qubit-pointer state Eq. 2.2.6.3 whose coherence decay, at short times, at the rate γ_m under photon loss.

Using the expressions Eq. 2.111 for the pointer states amplitude, γ_m can be expressed as

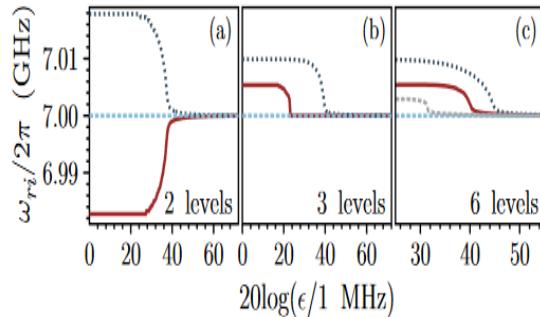
$$\gamma_m = \frac{\kappa\chi^2(\bar{n}_g + \bar{n}_e)}{\delta_r^2 + \chi^2 + (\kappa/2)^2} \quad (2.129)$$

with $\bar{n}_\sigma = |\alpha_\sigma|^2$ the average cavity photon number given that the qubit is state σ . The distinction between \bar{n}_g and \bar{n}_e is important if the measurement drive is not symmetrically placed between the two pulled cavity frequencies corresponding to the two qubit states. Taking $\delta_r = w_r - w_d = 0$ and thus $\bar{n}_g = \bar{n}_e \equiv \bar{n}$ for two-level system, the measurement-induced dephasing rate takes, in the small χ/κ limit, the simple form $\gamma_m \sim 8\chi\bar{n}/\kappa$. Thus as announced above, the qubit linewidth scales with \bar{n} . With the cautionary remarks that will come below, measuring this line width versus the drive power is thus another way to infer \bar{n} experimentally.

So far, we have been concerned with the small χ/κ limit. However, given the strong coupling and high-quality factor that can be experimentally realized in circuit QED, it is also interesting to consider the opposite limit where χ/κ is large. A first consequence of this strong dispersive regime, illustrated in Fig. 2.19 (b), is that the qubit frequency shift for per photon can then be large enough to be resolved spectroscopically. More precisely, this occurs if 2χ is larger than $\gamma_2 + (\bar{n} + n)\kappa/2$, the width of the n th photon peak. Moreover, the amplitude of each spectroscopic line is a measure of the probability of finding the corresponding photon

number in the cavity. Using this idea, it is possible, for example, to experimentally distinguish between coherent and thermal population of the cavity. This strong dependence of the qubit frequency on the exact photon number also allows for conditional qubit-cavity logical operations where, for example, a microwave pulse is applied such that qubit state is flipped if and only if there are n photons in the cavity. Although challenging, this strong dispersive limit has also been achieved in some cavity QED experiments. This regime has also been achieved in hybrid quantum systems, for example in phonon-number resolving measurements of nano mechanical oscillators and magnon-number resolving measurements.

We now come back to the question of inferring the intra-cavity photon number from ac-Stark shift or qubit linewidth broadening measurements. As mentioned previously, the linear dependence of the ac-Stark shift on the measurement drive power predicted from the dispersive Hamiltonian Eq. 2.109 is only valid at small \bar{n}/n_{crit} . Indeed, because of higher-order corrections, the cavity pull itself is not constant with \bar{n} but rather decreases with increasing \bar{n} . This change in cavity pull is illustrated in Fig. 2.20(a) which shows the effective resonator frequency given that the qubit is in state σ as a function of drive amplitude, $\omega_{r\sigma}(n) = E_{\sigma,n+1} - E_{\sigma,n}$, with $E_{\sigma,n+1}$ the dressed state energies defined in Eq. 2.26. At very low drive amplitude, the cavity frequency is pulled to the expected value $\omega_r \pm \chi$ depending on the state of the qubit. As the drive amplitude increases, and with it the intra-cavity photon number, the pulled cavity frequency goes back to its bare value ω_r . Panels (b) and (c) show the pulled frequencies taking into account three and six transmon levels, respectively. In contrast to the two-level approximation and as expected from Eq. 2.29, in this many-level situation the symmetry that was present in the two-level case is broken and the pulled frequencies are not symmetrically placed around ω_r . We note that this change in effective cavity frequency is at the heart of the high-power readout already discussed in signal-to-noise ratio and measurement fidelity section.



Because of this change in cavity pull, which can be interpreted as χ itself changing with photon numbers, the ac-Stark shift and the measurement-induced dephasing do not necessarily follow the simple linear dependence expected from \hat{H}_{disp} . For this reason, it is only possible to safely infer the intra-cavity photon number from measurement of the ac-Stark shift or qubit linewidth broadening at small photon number. It is worth noting that, in some cases, the reduction in cavity pull can move the cavity frequency closer to the drive frequency, thereby leading to an increase in cavity population. For some system parameters,

these two nonlinear effects – reduction in cavity pull and increase in cavity population – can partly compensate each other, leading to an *apparent* linear dependence of the qubit ac-Stark with power. We can only repeat that care must be taken when extracting the intra-cavity photon number in the dispersive regime.

2.2.7.3 Beyond strong coupling: ultrastrong coupling regime

We have discussed consequences of the strong coupling, $g > \kappa, \gamma_2$, and strong dispersive, $\chi > \kappa, \gamma_2$, regimes which can both easily be realized in circuit QED. Although the effect of light-matter interaction has important consequences, in both these regimes g is small with respect to the system frequencies, $\omega_r, \omega_q \gg g$, a fact that allowed us to safely drop counter-rotating terms from Eq. 2.18. In the case of a two-level system this allowed us to work with Jaynes-Cummings Hamiltonian Eq. 2.22. The situation where these terms can no longer be neglected is known as the ultrastrong coupling regime.

As discussed in exchange interaction between a transmon and an oscillator section, the relative smallness of g with respect to the system frequencies can be traced to Eq. 2.21 where we see that $g/\omega_r \propto \sqrt{\alpha}$, with $\alpha \sim 1/137$ the fine-structure constant. This is, however, not a fundamental limit and it is possible to take advantage of the flexibility of superconducting quantum circuits to engineer situations where light-matter coupling rather scales as $\propto 1/\sqrt{\alpha}$. In this case, the smallness of α now helps boost the coupling rather than constraining it. A circuit realizing this idea is commonly known as the in-line transmon. It simply consists of a transmission line resonator whose center conductor is interrupted by a Josephson junction. Coupling strength has large as $g/\omega_r \sim 0.15$ can in principle be obtained in this way but increasing this ratio further can be challenging because it is done at the expense of reducing the transmon anharmonicity.

An alternative approach relies on galvanically coupling a flux qubit to the center conductor of a transmission-line resonator. In this configuration, light-matter coupling can be made very large by increasing the impedance of the center conductor of the resonator in the vicinity of the qubit, something that can be realized by interrupting the center conductor of the resonator by a Josephson junction or a junction array. In this way, coupling strengths of $g/\omega_q \sim 1$ or larger can be achieved. These ideas were realized with $g/\omega_q \sim 0.1$ and more recently with coupling strengths as large as $g/\omega_q \sim 1.34$. Similar results have also been obtained in the context of waveguide QED where the qubit is coupled to an open transmission line rather than to a localized cavity mode.

A first consequence of reaching this ultrastrong coupling regime is that, in addition to a Lamb shift g^2/Δ , the qubit transition frequency is further modified by the so-called Bloch-Siegert shift of magnitude $g^2/(\omega_q + \omega_r)$. Another consequence is that the ground state of the combined system is no longer the factorizable state $|g0\rangle$ but is rather than an entangled qubit-resonator state. An immediate implication of this observation is that the master equation Eq. 2.70, whose steady-state is $|g0\rangle$, is not an appropriate description of damping in the ultrastrong coupling regime.

2.2.8 Quantum Computing with Circuit QED

One of the reasons for the rapid growth of circuit QED as a field of research is its prominent role in quantum computing. The transmon is today the most widely used superconducting qubit, and the dispersive measurement described in measurements in circuit QED section is the standard approach to qubit readout. Moreover, the capacitive coupling between transmons that are fabricated in close proximity can be used to implement two-qubit gates. Alternatively, the transmon-resonator electric dipole interaction can also be used to implement such gates between qubits that are separated by distances as large as a centimeter, the resonator acting as a quantum bus to mediate qubit-qubit interactions. Realizing a quantum computer architecture, even of modest size, required bringing together in a single working package essentially all of the elements discussed in this review.

In this section, we describe the basic principles behind one and two qubit gates in circuit QED. Our objective is not to give a complete overview of the many different gates and gate-optimization techniques that have been developed. We rather focus on the key aspects of how light-matter interaction facilitates coherent quantum operations for superconducting qubits, and describe some of the more commonly used gates to illustrate the basic principles. Unless otherwise noted, in this section we will assume the qubits to be dispersively coupled to the resonator.

2.2.8.1 Single-qubit control

Arbitrary single-qubit rotations can be realized in an NMR-like fashion with voltage drives at the qubit frequency. One approach is to drive the qubit via one of the resonator ports. Because of the large qubit-resonator detuning, a large fraction of the input power is reflected at the resonator, something that can be compensated by increasing the power emitted by the source. The reader will recognize that this approach is very similar to a qubit measurement but, because of the very large detuning, with $\delta_r \gg \chi$ such that $\alpha_e - \alpha_g \sim 0$ according to Eq. 2.112. As illustrated in Fig. 2.13, this far off-resonance drive therefore causes negligible measurement-induced dephasing. We also note that in the presence of multiple qubits coupled to the same resonator, it is important that the qubits be sufficiently detuned in frequency from each other to avoid the control drive intended for one qubit to inadvertently affect the other qubits.

Given this last constraint, a more convenient approach, already illustrated in Fig. 2.7, is to capacitively couple the qubit to an additional transmission line from which the control drives are applied. Of course, the coupling to this additional control port must be small enough to avoid any impact on the qubit relaxation time. Following controlling quantum systems with microwave drives section, the amplitude of the drive as seen by the qubit is given by $\epsilon = -i\sqrt{\gamma}\beta$, where β is the amplitude of the drive at the input port, and γ is set by the capacitance between the qubit and the transmission line. A small γ , corresponding to a long relaxation time, can be compensated by increasing the drive amplitude $|\beta|$, while making sure that any heating due to power dissipation close to the qubit does not affect qubit coherence. Design guidelines for wiring, an overview of the power dissipation induced by drive fields in qubit drive lines, and their effect on qubit coherence is discussed.

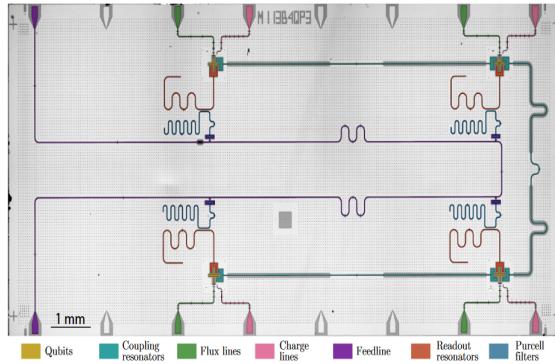
Similarly to Eq. 2.85, a coherent drive of time-dependent amplitude $\epsilon(t)$, frequency ω_d and phase ϕ_d on a transmon is then modeled by

$$\hat{H}(t) = \hat{H}_q + \hbar\epsilon(t) (\hat{b}^\dagger e^{-i\omega_d t - i\phi_d} + \hat{b} e^{i\omega_d t + i\phi_d}) \quad (2.130)$$

where $H_q = \hbar\omega_q \hat{b}^\dagger \hat{b} - \frac{E_C}{2} (\hat{b}^\dagger)^2 \hat{b}^2$ is the transmon Hamiltonian. Going to a frame rotating at ω_d , $\hat{H}(t)$ takes the simpler form

$$\hat{H}' = \hat{H}'_q + \hbar\epsilon(t) (\hat{b}^\dagger e^{-i\phi_d} + \hat{b} e^{i\phi_d}) \quad (2.131)$$

where $\hat{H}'_q = \hbar\delta_q \hat{b}^\dagger \hat{b} - \frac{E_c}{2} (\hat{b}^\dagger)^2 \hat{b}^2$ with $\delta_q = \omega_q - \omega_d$ the detuning between the qubit and the drive frequencies.



Truncating to two levels of the transmon as in 2.22, \hat{H}' takes the form

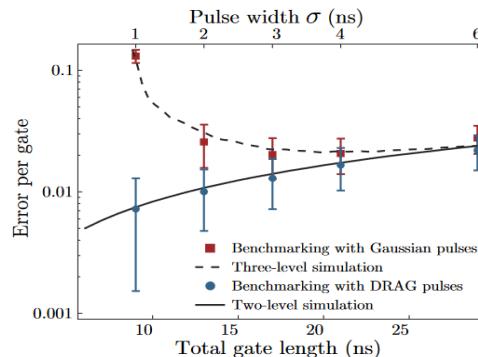
$$\hat{H}' = \frac{\hbar\delta_q}{2} \hat{\sigma}_z + \frac{\hbar\Omega_R(t)}{2} [\cos(\phi_d) \hat{\sigma}_x + \sin(\phi_d) \hat{\sigma}_y] \quad (2.132)$$

where we have introduced the standard notation $\Omega_R = 2\epsilon$ for the Rabi frequency. This form of \hat{H}' makes it clear how the phase of the drive, ϕ_d , controls the axis of rotation on the qubit Bloch sphere. Indeed, for $\delta_q = 0$, the choice $\phi_d = 0$ leads to rotations around the X-axis while $\phi_d = \pi/2$ to rotations around Y-axis. Since any rotation on the Bloch sphere can be decomposed into X and Y rotations, arbitrary single-qubit control is therefore possible using sequences of on-resonant drives with appropriate phases.

Implementing a desired gate requires turning on and off the drive amplitude. To realize as many logical operations as possible within the qubit coherence time, the gate time should be as short as possible and square pulses are optimal from that point of view. In practice, however, such pulses suffer from important deformation as they propagate down the finite-bandwidth transmission line from the source to the qubit. Moreover, for a weakly anharmonic multi-level system such as a transmon, high-frequency components of the square pulse can cause unwanted transitions to levels outside the two-level computational subspace. This

leakage can be avoided by using smooth pulses, but this leads to longer gate times. Another solution is to shape the pulse so as to remove the unwanted frequency components. A widely used approach that achieves this is known as Derivative Removal by Adiabatic Gate (DRAG). It is based on driving the two quadratures of the qubit with the envelope of the second quadrature chosen to be the time-derivative of the envelope of the first quadrature. More generally, one can cast the problem of finding an optimal drive as a numerical optimization problem which can be tackled with optimal control approaches such as the Gradient Ascent Pulse Engineering (GRAPE).

Experimental results comparing the error in single-qubit gates with and without DRAG are shown in Fig. 2.22. At long gate times, decoherence is the dominant source of error such that both Gaussian and DRAG pulses initially improve as the gate time is reduced. However, as the pulses get shorter and their frequency bandwidth become comparable to the transmon anharmonicity, leakage leads to large errors for the Gaussian pulses. In contrast, the DRAG results continue to improve as gates are made shorter and are consistent with a two-level system model of the transmon. These observations show that small anharmonicity is not a fundamental obstacle to fast and high-fidelity single-qubit gates. Indeed, thanks to pulse shaping techniques and long coherence times, state of the art single-qubit gate errors are below 10^{-3} , well below the threshold for topological error correcting codes.



While rotations about the Z axis can be realized by concentrating the X and Y rotations described above, several other approaches are used experimentally. Working in a rotating frame as in Eq. 2.132 with $\delta_q = 0$, one alternative method relies on changing the qubit transition frequency such that $\delta_q \neq 0$ for a determined duration. In the absence of drive, $\Omega_R = 0$, this leads to phase accumulation by the qubit state and therefore to a rotation about the Z axis. As discussed in flux-tunable transmons, fast changes of the qubit transition frequency are possible by, for example, applying a magnetic field to a flux-tunable transmon. However, working with flux-tunable transmons is done at cost of making the qubit susceptible to dephasing due to flux noise. To avoid this, the qubit transition frequency can also be tuned without relying on a flux-tunable device by applying a strongly detuned microwave tone on the qubit. For $\Omega_R/\delta_q \ll 1$, this drive does not lead to Rabi oscillations but induces an ac-Stark shift of the qubit frequency due to virtual transition caused by the drive. Indeed, to second order in Ω_R/δ_q and assuming for simplicity a constant drive amplitude, this situation is described by the effective Hamiltonian

$$\hat{H}'' \simeq \frac{1}{2} \left(\hbar\omega_q - \frac{E_C}{2} \frac{\Omega_R^2}{\delta_q^2} \right) \hat{\sigma}_z \quad (2.133)$$

The last term can be turned on and off with the amplitude of the detuned microwave drive and can therefore be used to realize Z rotations.

Finally, since the X and Y axis in Eq. 2.132 are defined by the phase ϕ_d of the drive, a particularly simple approach to realize a Z gate is to add the desired phase offset to the drive fields of all subsequent X and Y rotations and two-qubit gates. This so-called virtual Z-gate can be especially useful if the computation is optimized to use a large number of Z-rotations.

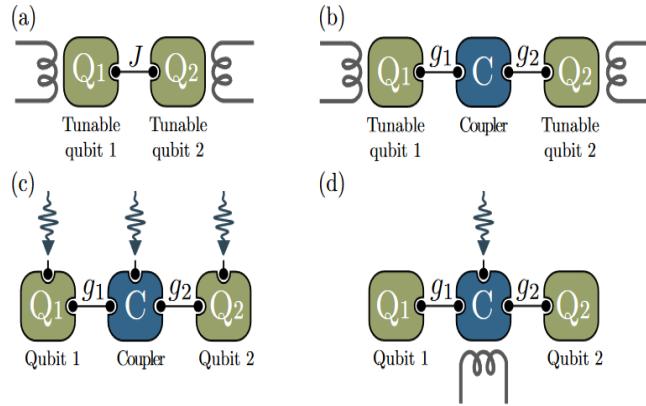
2.2.8.2 Two-qubit gates

Two-qubit gates are generally more challenging to realize than single-qubit gates. Error rates for current two-qubit gates are typically around one to a few percent, which is an order of magnitude higher than those of single-qubit gates. Recent experiments are, however, closing this gap. Improving two-qubit gate fidelities at short gate times is a very active area of research, and a wide variety of approaches have been developed. A key challenge in realizing two-qubit gates is the ability to rapidly turn interactions on and off. While for single-qubit gates this is done by simply turning on and off a microwave drive, two-qubit gates require turning on a coherent qubit-qubit interaction for a fixed time. Achieving large on/off ratios is far more challenging in this situation.

Broadly speaking, one can divide two-qubit gates into different categories depending on how the qubit-qubit interaction is activated. The main approaches discussed in the following are illustrated schematically in Fig. 2.23. An important distinction between these different schemes is whether they rely on frequency-tunable qubits or not. Frequency tunability is convenient because it can be used to controllably tune qubits into resonance with one another qubit or with a resonator. Using flux-tunable transmons has led to some of the fastest and highest fidelity two-qubit gates to date, see Fig. 2.23(a,b). However, as mentioned previously this leads to additional qubit dephasing due to flux noise. An alternative are all-microwave gates which only use microwave drives, either on the qubits or on a coupler bus such as a resonator, to activate an effective qubit-qubit interaction, see Fig. 2.23(c). Finally, yet another category of gates are parametric gates where a system parameter is modulated in time at a frequency which bridges an energy gap between the states of two qubits. Parametric gates can be all-microwave but, in some instances, involve modulating system frequencies using external magnetic flux, see Fig. 2.23(d).

1.Qubit-qubit exchange interaction

a. Direct capacitive coupling One of the concepually simplest ways to realize two-qubit gates is through direct capacitive coupling between the qubits, see Fig. 2.23(a). In analogy with Eq. 2.20, the Hamiltonian describing this situation reads



$$\hat{H} = \hat{H}_{q1} + \hat{H}_{q2} + \hbar J(\hat{b}_1^\dagger \hat{b}_2 + \hat{b}_1 \hat{b}_2^\dagger) \quad (2.134)$$

where $\hat{H}_{qi} = \hbar\omega_{qi}\hat{b}_i^\dagger \hat{b}_i - E_{Ci}(\hat{b}_i^\dagger)^2 \hat{b}_i^2 / 2$ is the Hamiltonian of the i th transmon and \hat{b}_i the corresponding annihilation operator. The interaction amplitude J takes the form

$$\hbar J = \frac{2E_{C1}E_{C2}}{E_{Cc}} \left(\frac{E_{J1}}{2E_{C1}} \times \frac{E_{J2}}{2E_{C2}} \right)^{1/4} \quad (2.135)$$

with E_{J_i} and E_{Ci} the transmon Josephson and charging energies, and $E_{Cc} = e^2/2C_c$ the charging energy of the coupling capacitance labelled C_c . This beam-splitter Hamiltonian describes the coherent exchange of an excitation between the two qubits. In the two-level approximation, assuming tuned to resonance qubit, $\omega_{q1} = \omega_{q2}$, and moving to a frame rotating at the qubit frequency, Eq.2.134 takes the familiar form,

$$\hat{H}' = \hbar J (\hat{\sigma}_{+1}\hat{\sigma}_{-2} + \hat{\sigma}_{-1}\hat{\sigma}_{+2}) \quad (2.136)$$

Evolution under this Hamiltonian for a time $\pi/(4J)$ leads to an entangling \sqrt{iSWAP} gate which, up to single-qubit rotations, is equivalent to a controlled NOT gate (CNOT).

As already mentioned, to precisely control the evolution under \hat{H}' and therefore the gate time, it is essential to be able to vary the qubit-qubit interaction with a large on/off ratio. There are essentially two approaches to realizing this. The most straightforward way is to tune the qubits in resonance to perform a two-qubit gate, and to strongly detune them to stop the coherent exchange induced by \hat{H}' . Indeed, for $J/\Delta_{12} \ll 1$ where $\Delta_{12} = \omega_{q1} - \omega_{q2}$ change J is suppressed and can be dropped from Eq. 2.134 under the RWA. A more careful analysis following the same arguments and approach used to describe the dispersive regime shows that, to second order in J/Δ_{12} , a residual qubit-qubit interaction of the form $(J^2/\Delta_{12})\hat{\sigma}_{z1}\hat{\sigma}_{z2}$ remains. This unwanted interaction in the off state of the gate leads to a conditional phase accumulation on the qubits. As a result, the on-off ratio of this direct coupling gate is estimated to be $\sim \Delta_{12}/J$. In practice, this ratio cannot be made arbitrarily

small because increasing the detuning of one pair of qubits in a multi-qubit architecture might lead to accidental resonance with a third qubit. This direct coupling approach was implemented using frequency tunable transmons with a coupling $J/2\pi = 30$ MHz and an on/off ratio of 100. We note that the unwanted phase accumulation due to the residual $\hat{\sigma}_{z1}\hat{\sigma}_{z2}$ can in principle be eliminated using refocusing techniques borrowed from nuclear magnetic resonance.

Another approach to turn on and off the swap interaction is to make the J coupling itself tunable in time. This is conceptually simple, but requires more complex coupling circuitry typically involving flux-tunable elements that can open additional decoherence channels for the qubits. One advantage is, however, that tuning a coupler rather than qubit transition frequencies helps in reducing the frequency crowding problem. This approach is used, for example, where two transmon qubits are coupled via a flux-tunable inductive coupler. In this way, it was possible to realize an on/off ratio of 1000, with a maximum coupling of 100 MHz corresponding to a \sqrt{iSWAP} gate in 2.5 ns. A simpler approach based on a frequency tunable transmon qubit acting as coupler, was also used to tune qubit-qubit coupling from 5 MHz to -40 MHz going through zero coupling with a gate time of ~ 12 ns and gate infidelity of $\sim 0.5\%$.

b. Resonator mediated coupling An alternative to the above approach is to use a resonator as a quantum bus mediating interactions between two qubits. An advantage compared to direct coupling is that the qubits do not have to be fabricated in close proximity to each other. With the qubits coupled to the same resonator, and in the absence of any direct coupling between the qubits, the Hamiltonian describing this situation is

$$\hat{H} = \hat{H}_{q1} + \hat{H}_{q2} + \hbar\omega_r \hat{a}^\dagger \hat{a} + \sum_{i=1}^2 \hbar g_i (\hat{a}^\dagger \hat{b}_i + \hat{a} \hat{b}_i^\dagger) \quad (2.137)$$

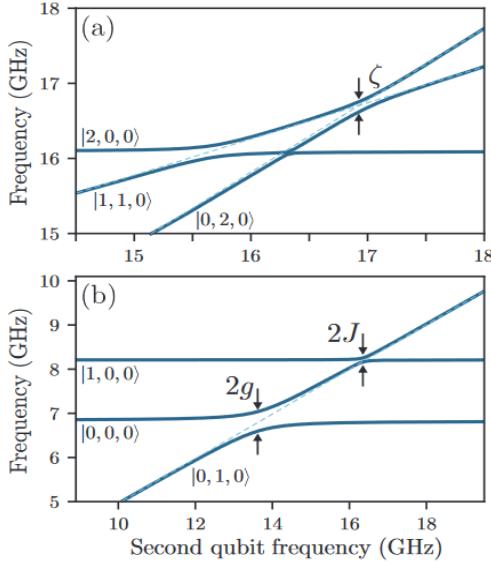
One way to make use of this pairwise interaction is, assuming the resonator to be in the vacuum state, to first tune one of the two qubits in resonance with the resonator for half a vacuum Rabi oscillation cycle, swapping an excitation from the qubit to the resonator, before tuning it back out of resonance. The second qubit is then tuned in resonance mapping the excitation from the resonator to the second qubit. While this sequence of operations can swap the quantum state of the first qubit to the second, clearly demonstrating the role of the resonator as a quantum bus, it does not correspond to an entangling two-qubit gate.

Alternatively, a two-qubit gate can be performed by only virtually populating the resonator mode by working in the dispersive regime where both qubits are far detuned from resonator. Building on the results of dispersive regime section, in this situation the effective qubit-qubit interaction is revealed by using the approximate dispersive transformation

$$\hat{U} = \exp \left[\sum_i \frac{g_i}{\Delta_i} (\hat{a}^\dagger \hat{b}_i - \hat{a} \hat{b}_i^\dagger) \right]$$

on Eq. 2.137. Making use of the Baker-Campbell-Hausdorff expansion Eq. to second order in g_i/Δ_i , we find

$$\begin{aligned}
\hat{H}' = & \hat{H}'_{q1} + \hat{H}'_{q2} + \hbar J \left(\hat{b}_1^\dagger \hat{b}_2 + \hat{b}_1 \hat{b}_2^\dagger \right) \\
& + \hbar \tilde{\omega}_r \hat{a}^\dagger \hat{a} + \sum_{i=1}^2 \hbar \chi_{ab_i} \hat{a}^\dagger \hat{a} \hat{b}_i^\dagger \hat{b}_i \\
& + \sum_{i \neq j} \hbar \Xi_{ij} \hat{b}_i^\dagger \hat{b}_i \left(\hat{b}_j^\dagger \hat{b}_i + \hat{b}_i^\dagger \hat{b}_j \right)
\end{aligned} \tag{2.138}$$



with $H'_{qi} \simeq \hbar \tilde{\omega}_{qi} \hat{b}_i^\dagger \hat{b}_i - \frac{E_{Ci}}{2} \left(\hat{b}_i^\dagger \right)^2 \hat{b}_i^2$ the transmon Hamiltonians, $\chi_{ab_i} \simeq -2E_{Ci}g_i^2/\Delta_i^2$ a cross-Kerr coupling between the resonator and the i th qubit. The frequencies $\tilde{\omega}_{qi}$ and $\tilde{\omega}_r$ include the Lamb shift. The last line can be understood as an excitation number dependent exchange interaction with $\Xi_{ij} = E_{Ci}g_i g_j/(2\Delta_i \Delta_j)$. Since this term is much smaller than the J -coupling it can typically be neglected. Note that we have not included a self-Kerr term of order χ_{ab_i} on the resonator. This term is of no practical consequences in the dispersive regime where the resonator is only virtually populated. The resonator-induced J coupling in \hat{H}' takes the form

$$J = \frac{g_1 g_2}{2} \left(\frac{1}{\Delta_1} + \frac{1}{\Delta_2} \right) \tag{2.139}$$

and reveals itself in the frequency domain by an anticrossing of size $2J$ between the qubit states $|01\rangle$ and $|10\rangle$. This is illustrated in Fig. 2.24 (b) which shows the eigenenergies of the Hamiltonian Eq. 2.137 in the 1-excitation manifold. In this figure, the frequency of qubit 1 is swept while that of qubit 2 is kept constant at ~ 8 GHz with the resonator at ~ 7 GHz. From left to right, we first see the vacuum Rabi splitting of size $2g$ at $\omega_{q1} = \omega_r$, followed by a smaller anticrossing of size $2J$ at the qubit-qubit resonance. It is worth mentioning that

the above expression for J is only valid for single-mode oscillators and is renormalized in the presence of multiple modes.

To understand the consequences of the J -coupling in the time domain, it is useful to note that, if the resonator is initially in the vacuum state, it will remain in that state under the influence of \hat{H}' . In other words, the resonator is only virtually populated by its dispersive interaction with the qubits. For this reason, with the resonator initialized in the vacuum state, the second line of Eq. 2.138 can for all practical purposes be ignored and we are back to the form of the direct coupling Hamiltonian of Eq. 2.134. Consequently, when both qubits are tuned in resonance with each other, but still dispersive with respect to the resonator, the latter acts as a quantum bus mediating interactions between the qubits. An entangling gate can thus be performed in the same way as with direct capacitive coupling, either by tuning the qubits in and out of resonance with each other or by making the couplings g_i tunable.

2. Flux-tuned 11-02 phase gate

The 11-02 phase gate is a controlled-phase gate that is well suited to weakly anharmonic qubits such as transmons. It is obtained from the exchange interaction of Eq. 2.134 and can thus be realized through direct (static or tunable) qubit-qubit coupling or indirect coupling via a resonator bus.

In contrast to the \sqrt{iSWAP} gate, the 11-02 phase gate is not based on tuning the qubit transition frequencies between the computational states into resonance with each other, but rather exploits the third energy levels of the transmon. The 11-02 gate relies on tuning the qubits to a point where the states $|11\rangle$ and $|02\rangle$ are degenerate in the absence of J coupling. As illustrated in Fig. 2.24(a), the qubit-qubit coupling lifts this degeneracy by an energy ζ whose value can be found perturbatively. Because of this repulsion caused by coupling to the state $|02\rangle$, the energy E_{11} of the state $|11\rangle$ is smaller than $E_{01} + E_{10}$ by ζ . Adiabatically flux tuning the qubits in and out of the 11-02 anticrossing therefore leads to a conditional phase accumulation which is equivalent to the controlled-phase gate.

To see this more clearly, it is useful to write the unitary corresponding to this adiabatic time evolution as

$$\hat{C}_Z(\phi_{01}, \phi_{10}, \phi_{11}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i\phi_{01}} & 0 & 0 \\ 0 & 0 & e^{i\phi_{10}} & 0 \\ 0 & 0 & 0 & e^{i\phi_{11}} \end{pmatrix} \quad (2.140)$$

where $\phi_{ab} = \int dt E_{ab}(t)/\hbar$ is the dynamical phase accumulated over total flux excursion. Up to single-qubit rotations, this is equivalent to a standard controlled-phase gate since

$$\begin{aligned}\hat{C}_Z(\phi) &= \text{diag}(1, 1, 1, e^{i\phi}) \\ &= \hat{R}_Z^1(-\phi_{10}) \hat{R}_Z^2(-\phi_{01}) \hat{C}_Z(\phi_{01}, \phi_{10}, \phi_{11})\end{aligned}\quad (2.141)$$

with $\phi = \phi_{11} - \phi_{01} - \phi_{10} = \int dt \zeta(t)$ and where $\hat{R}_Z^i(\theta) = \text{diag}(1, e^{i\theta})$ is a single qubit phase gate acting on qubit i . For $\phi \neq 0$ this is an entangling two-qubit gate and, in particular, for $\theta = \pi$ it is a controlled-Z gate (CPHASE).

Rather than adiabatically tuning the flux in and out of the 11 -02 resonance, an alternative is to non-adiabatically pulse to this anti-crossing. In this sudden approximation, leaving the system there for a time t , the state $|11\rangle$ evolves into $\cos(\zeta t/2\hbar)|11\rangle + \sin(\zeta t/2\hbar)|02\rangle$. For $t = \hbar/\zeta$, $|11\rangle$ is mapped back into itself but acquires a minus sign in the process. On the other hand, since they are far from any resonance, the other logical states evolve trivially. This therefore again results in a CPHASE gate. In this way, fast controlled-Z gates are possible. For direct qubit-qubit coupling in particular, some of the fastest and highest fidelity two-qubit gates have been achieved this way with error rates below the percent level and gate times of a few tens of ns.

Despite its advantages, a challenge associated with this gate is the distortions in the flux pulses due to the finite bandwidth of the control electronics and line. In addition to modifying the waveform experienced by the qubit, this can lead to long time scale distortions where the flux at the qubit at a given time depends on the previous flux excursions. This situation can be partially solved by pre-distorting the pulses taking into account the known distortion, but also by adapting the applied flux pulses to take advantage of the symmetry around the transmon sweet-spot to cancel out unwanted contributions.

3. All-microwave gates

Because the on/off ratio of the gates discussed above is controlled by the detuning between the qubits, it is necessary to tune the qubit frequencies over relatively large frequency ranges or, alternatively, to have tunable coupling elements. In both cases, having a handle on the qubit frequency or qubit-qubit coupling opens the system to additional dephasing. Moreover, changing the qubit frequency over large ranges can lead to accidental resonance with other qubits or uncontrolled environmental modes, resulting in energy loss. For these reasons, it can be advantageous to control two-qubit gates in very much the same way as single-qubit gates: by simply turning on and off a microwave drive. In this section, we describe two so-called all-microwave gates: the resonator induced phase (RIP) gate and the cross-resonance (CR) gate. Both are based on fixed-frequency far off-resonance qubits with an always-on qubit-resonator coupling. The RIP gate is activated by driving a common resonator and the CR gate by driving one of the qubits. Other all microwave gates which will not be discussed further here include the sideband-based iSWAP, the bSWAP, the microwave-activated CPHASE and fg-ge gate.

a. Resonator-induced phase gate The RIP gate relies on two strongly detuned qubits that are

dispersively coupled to a common resonator mode. The starting point is thus Eq. 2.138 where we now neglect the J coupling by taking $|\omega_{q1} - \omega_{q2}| \gg J$. In the two-level approximation and accounting for a drive on the resonator, this situation is described by the Hamiltonian

$$\begin{aligned}\hat{H}'/\hbar = & \frac{\tilde{\omega}_{q1}}{2}\hat{\sigma}_{z1} + \frac{\tilde{\omega}_{q2}}{2}\hat{\sigma}_{z2} + \omega\tilde{\omega}_r\hat{a}^\dagger\hat{a} \\ & + \sum_{i=1}^2 \chi_i\hat{a}^\dagger\hat{a}\hat{\sigma}_{zi} + \epsilon(t)(\hat{a}^\dagger e^{-i\omega_dt} + \hat{a}e^{i\omega_dt})\end{aligned}\quad (2.142)$$

where $\epsilon(t)$ is the time-dependent amplitude of the resonator drive and ω_d its frequency. Note that we also neglect the resonator self-Kerr nonlinearity.

The gate is realized by adiabatically ramping on and off the drive $\epsilon(t)$, such that the resonator starts and ends in the vacuum state. Crucially, this means that the resonator is unentangled from the qubits at the start and end of the gate. Moreover, to avoid measurement-induced dephasing, the drive frequency is chosen to be far from the cavity mode, $\tilde{\delta}_r = \tilde{\omega}_r - \tilde{\omega}_d \gg \kappa$. Despite this strong detuning, the dispersive shift causes the resonator frequency to depend on the state of the two qubits and, as a result, the resonator field evolves in a closed path in phase space that is qubit-state dependent. This leads to a different phase accumulation for the different qubit states, and therefore to a controlled-phase gate of the form of Eq. 2.140.

This conditional phase accumulation can be made more apparent by moving Eq. 2.142 to a frame rotating at the drive frequency and by applying the polaron transformation $\hat{U} = \exp[\hat{a}'(t)\hat{a}^\dagger - \hat{a}^{*\prime}(t)\hat{a}]$ with $\alpha'(t) = \alpha(t) - \sum_i \chi_i \hat{\sigma}_{zi}/\tilde{\delta}_r$ on the resulting Hamiltonian. This leads to the approximate effective Hamiltonian

$$\begin{aligned}\hat{H}'' \simeq & \sum_i \hbar \left[\frac{\tilde{\delta}_{qi}}{2} + \chi |\alpha(t)|^2 \right] \hat{\sigma}_{zi} + \hbar \delta_r \hat{a}^\dagger \hat{a} \\ & + \sum_{i=1}^2 \hbar \chi_i \hat{a}^\dagger \hat{a} \hat{\sigma}_{zi} - \hbar \frac{2\chi_1 \chi_2 |\alpha(t)|^2}{\delta_r} \hat{\sigma}_{z1} \hat{\sigma}_{z2}\end{aligned}\quad (2.143)$$

with $\tilde{\delta}_x = \tilde{\omega}_x - \omega_d$ and where the field amplitude $\alpha(t)$ satisfies $\dot{\alpha} = -i\tilde{\delta}_r\alpha - i\epsilon(t)$. In this frame, it is clear how the resonator mediates a $\hat{\sigma}_{z1}\hat{\sigma}_{z2}$ interaction between the two qubits and therefore leads to a conditional phase gate. This expression also makes it clear that the need to avoid measurement-induced dephasing with $\tilde{\delta}_r \gg \kappa$ limits the effective interaction strength and therefore leads to relatively long gate times. This can, however, be mitigated by taking advantage of pulse shaping techniques or by using squeezed radiation to erase the which-qubit information in the output field of the resonator. Similarly to the longitudinal readout protocol, longitudinal coupling also offers a way to overcome many of the limitations of the conventional RIP gate.

Some of the advantages of this two-qubit gate are that it can couple qubits that are far detuned from each other and that it does not introduce significant leakage errors. This gate

was demonstrated with multiple transmons coupled to a 3D resonator, achieving error rates of a few percent and gate times of several hundred nanoseconds.

b. Cross-resonance gate The cross-resonance gate is based on qubits that are detuned from each other and coupled by an exchange term J of the form of Eq. 2.134 or Eq. 2.138. While the RIP gate relies on off-resonant driving of a common oscillator mode, this gate is based on directly driving one of the qubits at the frequency of the other. Moreover, since the resonator is not directly used and, in fact, ideally remains in its vacuum throughout the gate, the J coupling can be mediated by a resonator or by direct capacitive coupling.

In the two-level approximation and in the absence of the drive, this interaction takes the form

$$\hat{H} = \frac{\hbar\omega_{q1}}{2}\hat{\sigma}_{z1} + \frac{\hbar\omega_{q2}}{2}\hat{\sigma}_{z2} + \hbar J(\hat{\sigma}_{+1}\hat{\sigma}_{-2} + \hat{\sigma}_{-1}\hat{\sigma}_{+2}) \quad (2.144)$$

To see how this gate operates, it is useful to diagonalize \hat{H} using the two-level system version of the transformation Eq. 2.36. The result takes the same general form as Eq. 2.37 and Eq. 2.38, after projecting to two levels. In this frame, the presence of the J coupling leads to a renormalization of the qubit frequencies which for strongly detuned qubits, $|\Delta_{12}| = |\omega_{q1} * \omega_{q2}| \gg |J|$, take the values $\tilde{\omega}_{q1} \approx \omega_{q1} + J^2/\Delta_{12}$ to second order in J/Δ_{12} . In the same frame, a drive on the first qubit, $\hbar\Omega_R(t)\cos(\omega_d t)\hat{\sigma}_{x1}$, takes the form

$$\begin{aligned} & \hbar\Omega_R(t)\cos(\omega_d t)(\cos\theta\hat{\sigma}_{x1} + \sin\theta\hat{\sigma}_{z1}\hat{\sigma}_{x2}) \\ & \approx \hbar\Omega_R(t)\cos(\omega_d t)\left(\hat{\sigma}_{x1} + \frac{J}{\Delta_{12}}\hat{\sigma}_{z1}\hat{\sigma}_{x2}\right) \end{aligned} \quad (2.145)$$

with $\theta = \arctan(2J/\Delta_{12})/2$ and where the second line is valid to first order in J/Δ_{12} . As a result, driving the first qubit at the frequency of the second qubit, $\omega_d = \tilde{\omega}_{q2}$, activates the term $\hat{\sigma}_{z1}\hat{\sigma}_{x2}$ which can be used to realize a CNOT gate.

More accurate expressions for the amplitude of the CR term $\hat{\sigma}_{z1}\hat{\sigma}_{x2}$ can be obtained by taking into account more levels of the transmons. In this case, the starting point is the Hamiltonian Eq. 2.134 with, as above, a drive term on the first qubit

$$\begin{aligned} \hat{H} = & \hat{H}_{q1} + \hat{H}_{q2} + \hbar J\left(\hat{b}_1^\dagger\hat{b}_2 + \hat{b}_1\hat{b}_2^\dagger\right) \\ & + \hbar\varepsilon(t)\left(\hat{b}_1^\dagger e^{-i\omega_d t} + \hat{b}_1 e^{i\omega_d t}\right) \end{aligned} \quad (2.146)$$

where $\omega_d \sim \omega_{q2}$. Similarly to the previous two-level system example, it is useful to eliminate the J -coupling. We do this by moving to a rotating frame at the drive frequency for both qubits, followed by a Schrieffer-Wolff transformation to diagonalize the first line of the equation to second order in J . The drive term is modified under the same transformation by using the explicit expression for the Schrieffer-Wolff generator $\hat{S} = \hat{S}^{(1)} + \dots$, and the Baker-Campbell-Hausdorff formula to first order: $e^{\hat{S}}\hat{b}_1e^{-\hat{S}} \simeq \hat{b}_1 + [\hat{S}^{(1)}, \hat{b}_1]$. The full calculation is fairly involved and here we only quote the final result after truncating to the two lowest levels of the transmon qubits.

$$\hat{H}' \simeq \frac{\hbar\tilde{\delta}_{q1}}{2}\hat{\sigma}_{z1} + \frac{\hbar\hat{\delta}_{q2}}{2}\hat{\sigma}_{z2} + \frac{\hbar\chi_{12}}{2}\hat{\sigma}_{z1}\hat{\sigma}_{z2} + \hbar\varepsilon(t) \left(\hat{\sigma}_{x1} - J'\hat{\sigma}_{x2} - \frac{E_{C1}}{\hbar} \frac{J'}{\Delta_{12}} \hat{\sigma}_{z1}\hat{\sigma}_{x2} \right) \quad (2.147)$$

In this expression, the detunings include frequency shifts due to the J coupling with $\tilde{\delta}_{q1} = \omega_{q1} + J^2/\Delta_{12} + \chi_{12} - \omega_d$ and $\tilde{\delta}_{q2} = \omega_2 - J^2/\Delta_{12} + \chi_{12} - \omega_d$. The parameters χ_{12} and J' are given by

$$\begin{aligned} \chi_{12} &= \frac{J^2}{\Delta_{12} + \frac{E_{C2}}{\hbar}} - \frac{J^2}{\Delta_{12} - \frac{E_{C1}}{\hbar}} \\ J' &= \frac{J}{\Delta_{12} - \frac{E_{C1}}{\hbar}} \end{aligned} \quad (2.148)$$

Equations 2.145 and 2.147 agree in the limit of large anharmonicity $E_{C1,2}$ and we again find that a drive on the first qubit at the frequency of the second qubit activates the CR term $\hat{\sigma}_{z1}\hat{\sigma}_{x2}$. However, there are important differences at finite $E_{C1,2}$, something which highlights the importance of taking into account the multilevel nature of the transmon. Indeed, the amplitude of the Cr term is smaller here than in Eq. 2.145 with a two-level system. Moreover, in constant to the latter case, when taking into account multiple levels of the transmon qubits we find a spurious interaction $\hat{\sigma}_{z1}\hat{\sigma}_{z2}$ of amplitude χ_{12} between the two qubits, as well as a drive on the second qubit of amplitude $J'\epsilon(t)$. This unwanted drive can be echoed away with additional single-qubit gates. The $\hat{\sigma}_{z1}\hat{\sigma}_{z2}$ interaction is detrimental to the gate fidelity as it effectively makes the frequency of the second qubit dependent on the logical state of the first qubit. Because of this, the effective dressed frequency of the second qubit cannot be known in general, such that it is not possible to choose the drive frequency ω_d to be on resonance with the second qubit, irrespective of the state of the first. As a consequence, the CR term $\hat{\sigma}_{z1}\hat{\sigma}_{x2}$ in Eq. 2.147 will rotate at an unknown qubit-state dependent frequency, leading to a gate error. The $\hat{\sigma}_{z1}\hat{\sigma}_{x2}$ term should therefore be made small, which ultimately limits the gate speed. Interestingly, for a pair of qubits with equal and opposite anharmonicity, $\chi_{12} = 0$ and this unwanted effect is absent. This cannot be realized with two conventional transmons, but is possible with other types of qubits.

Since J' is small, another caveat of the CR gate is that large microwave amplitudes ϵ are required for fast gates. For the typical low-anharmonicity of transmon qubits, this can lead to leakages and to effects that are not captured by the second-order perturbative results of Eqs. 2.145 and 2.147. More detailed modelling based on the Hamiltonian of Eq. 2.146 suggests that classical crosstalk induced on the second qubit from driving the first qubit can be important and is a source of discrepancy between the simple two-level system model and experiments. Because of these spurious effects, CR gate times have typically been relatively long, of the order of 300 to 400 ns with gate fidelities $\sim 94 - 96\%$. However, with careful calibration and modelling beyond Eq. 2.147, it has been possible to push gate times down to the 100 – 200 ns range with error per gates at the percent level.

Similarly to RIP gate, advantages of the CR gate include the fact that realizing this gate can be realized using the same drive lines that are used for single-qubit gates. Moreover, it works with fixed frequency qubit gates. Moreover, it works with fixed frequency qubits which often have longer phase coherence times than their flux-tunable counterparts. However, both the RIP and the CR gate are slowe than what can now be achieved with additional flux control of the qubit frequency or of the coupler. We also note that, due to the factor $E_{C1}/\hbar\Delta_{12}$ in the amplitude of the $\hat{\sigma}_z^1\hat{\sigma}_z^2$ term, the detuning of the two qubits cannot be too large compared to the anharmonicity, putting further constraints on the choice of the qubit frequencies. This may lead to frequency crowding issues when working with large numbers of qubits.

4. Parametric gates

As we have already discussed, a challenge in realizing two-qubit gates is activating a coherent interaction between two qubits with a large on/off ratio. The gates discussed so far have aimed to achieve this in different ways. The \sqrt{iSWAP} and the 11-02 gates are based on flux-tuning qubits into a resonance condition or on a tunable coupling element. The RIP gate is based on activating an effective qubit-qubit coupling by driving a resonator and the CR gate by driving one of the qubits. Another approach is to activate an off-resonant interaction by modulating a qubit frequency, a resonator frequency, or the coupling parameter itself at an appropriate frequency. This parametric modulation provides the energy necessary to bridge the energy gap between the far detuned qubit states. Several such schemes, known as parametric gates, have been theoretically developed and experimentally realized.

The key idea behind parametric gates is that modulation of a system parameter can induce transitions between energy levels that would otherwise be too far off-resonance to give any appreciable coupling. We illustrate the idea first with two directly coupled qubits described by the Hamiltonian

$$\hat{H} = \frac{\hbar\omega_{q1}}{2}\hat{\sigma}_{z1} + \frac{\hbar\omega_{q2}}{2}\hat{\sigma}_{z2} + J(t)\hat{\sigma}_{x1}\hat{\sigma}_{x2} \quad (2.149)$$

where we assume that the coupling is periodically modulated at the frequency ω_m , $J(t) = J_0 + \tilde{J}\cos(\omega_mt)$. Moving to a rotating frame at the qubit frequencies, the above Hamiltonian takes the form

$$\begin{aligned} \hat{H}' - J(t) & \left(e^{i(\omega_{q1}-\omega_{q2})t} \hat{\sigma}_{+1}\hat{\sigma}_{-2} - 2 \right. \\ & \left. + e^{i(\omega_{q1}+\omega_{q2})t} \hat{\sigma}_{+1}\hat{\sigma}_{+2} + H.c. \right) \end{aligned} \quad (2.150)$$

Just as in qubit-qubit exchange interaction section, if the coupling is constant $J(t) = J_0$, and $J_0/(\omega_{q1} - \omega_{q2})$, $J_0/(\omega_{q1} + \omega_{q2} \ll 1)$, then all the terms of \hat{H}' are fast-rotating and can be neglected. In this situation, the gate is in the off state. On the other hand, by appropriately choosing the modulation frequency ω_m , it is possible to selectively activate some of these

term. Indeed, for $\omega_m = \omega_{q1} - \omega_{q2}$, the terms $\hat{\sigma}_+ \hat{\omega}_{-2} + H.c.$ are no longer rotating and are effectively resonant. Dropping the rapidly rotating terms, this leads to

$$\hat{H}' \simeq \frac{\tilde{J}}{2} (\hat{\sigma} + 1\hat{\sigma} - 2 + \hat{\sigma}_{-1}\hat{\sigma} + 2). \quad (2.151)$$

As already discussed, this interaction can be used to generate entangling gates such as the \sqrt{iSWAP} . If rather $\omega_m = \omega_1 + \omega_2$ then $\hat{\sigma}_{+1}\hat{\sigma}_{+2} + H.c.$ is instead selected.

In practice, it can sometimes be easier to modulate a qubit or resonator frequency rather than a coupling strength. To see how this leads to a similar result, consider the Hamiltonian

$$\hat{H} = \frac{\hbar\omega_{q1}(t)}{2} \hat{\sigma}_{z1} + \frac{\hbar\omega_{q2}}{2} \hat{\sigma}_{z2} + J \hat{\sigma}_{x1} \hat{\sigma}_{x2} \quad (2.152)$$

Taking $\omega_{q1}(t) = \omega_{q1} + \epsilon \sin(\omega_m t)$, the transition frequency of the first qubit develops frequency modulation (FM) sidebands. The two qubits can then be effectively brought into resonance by choosing the modulation to align one of the FM sidebands with ω_{q2} , thereby, rendering the J effectively coupling resonant. This can be seen more clearly by moving to a rotating frame defined by the unitary

$$\hat{U} = e^{-\frac{i}{2} \int_0^t dt' \omega_{q1}(t') \sigma z_1} e^{-i\omega_{q2}t \hat{\sigma}_{z2}/2} \quad (2.153)$$

where the Hamiltonian takes the form

$$\begin{aligned} \hat{H}' = & J \sum_{n=-\infty}^{\infty} J_n \left(\frac{\epsilon}{\omega_m} \right) (i^n e^{i(\Delta_{12} - n\omega_m)t} \hat{\sigma}_{+1} \hat{\sigma}_{-2} \\ & + i^n e^{i(\omega_{q1} + \omega_{q2} - n\omega_m t)t} \hat{\sigma}_{+1} \hat{\sigma}_{+2} + H.c.) \end{aligned} \quad (2.154)$$

To arrive at the above expression, we have used the Jacobi-Anger expansion $e^{iz\cos\theta} = \sum_{n=-\infty}^{\infty} i^n J_n(z) e^{in\theta}$, with modulation frequency such that $n\omega_m = \Delta_{12}$ aligns the n th sideband with the resonator frequency such that a resonant qubit-resonator interaction is recovered. The largest contribution comes from the first sideband with J_1 which has a maximum around $J_1(1.84) \simeq 0.58$, thus corresponding to an effective coupling that is a large fraction of the bare J coupling. Note that the assumption of having a simple sinusoidal modulation of frequency neglects the fact that the qubit frequency has a nonlinear dependence on external flux for tunable transmons. This behaviour can still be approximated by appropriately, varying $\Phi_x(t)$.

Parametric gates can also be mediated by modulating the frequency of a resonator bus to which qubits are dispersively coupled. Much as with flux-tunable transmons, the resonator is made tunable by inserting a SQUID loop in the center conductor of the resonator. Changing the flux threading the SQUID loop changes the SQUID's inductance and therefore the

effective length of the resonator. As in trombone, this leads to a change of the resonator frequency. An advantage of modulating the resonator bus over modulating the qubit frequency is that the latter can have a fixed frequency, thus reducing its susceptibility to flux noise.

Finally, it is worth pointing out that while the speed of the cross-resonance gate is reduced when the qubit-qubit detuning is larger than the transmon anharmonicity, parametric gates do not suffer from this problem. As a result, there is more freedom in the choice of the qubit frequencies with parametric gates, which is advantageous to avoid frequency crowding related issues such as addressability errors and crosstalk. We also note that the modulation frequencies required to activate parametric gates can be a few hundred MHz, in contrast to the RIP gate or the CR gate which require microwave drives. Removing the need for additional microwave generators simplifies the control electronics and *may* help make the process more scalable. A counterpoint is that fast parametric gates often require large modulation amplitudes, which can be challenging.

2.2.8.3 C. Encoding a qubit in an oscillator

So far we have discussed encoding of quantum information into the first two energy levels of an artificial atom, the cavity being used for readout and two-qubit gates. However, cavity modes often have superior coherence properties than superconducting artificial atoms, something that is especially true for the 3D cavities discussed in 3D resonator section. This suggests that encoding quantum information in the oscillator mode can be advantageous. Using oscillator modes to store and manipulate quantum information can also be favorable for quantum error correction which is an essential aspect of scalable quantum computer architectures.

Indeed, in addition to their long coherence time, oscillators have a simple and relatively well-understood error model: to a large extent, the dominant error is single-photon loss. Taking advantage of this, it is possible to design quantum error correction codes that specifically correct for this most likely error. This is to be contrasted to more standard codes, such as the surface code, which aim at detecting and correcting both amplitude and phase errors. Moreover, as will become clear below, the infinite dimensional Hilbert space of a single oscillator can be exploited to provide the redundancy which is necessary for error correction thereby, in principle, allowing using less physical resources to protect quantum information than when using two-level systems. Finally, qubits encoded in oscillators can be concatenated with conventional error correcting codes, where the latter should be optimized to exploit the noise resilience provided by the oscillator encoding.

Of course, as we have already argued, nonlinearity remains essential to prepare and manipulate quantum states of the oscillator. When encoding quantum information in a cavity mode, a dispersively coupled artificial atom (or other Josephson junction-based circuit element) remains present but only to provide nonlinearity to the oscillator ideally without playing much of an active role.

Oscillator encodings of qubits investigated in the context of quantum optics and circuit QED include cat codes, as well as a two-mode amplitude damping code.

To understand the basic idea behind this approach, we first consider the simplest instance of the binomial code in which a qubit is encoded in the following two states of a resonator mode

$$|0_L\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |4\rangle), \quad |1_L\rangle = |2\rangle \quad (2.155)$$

with Fock states $|n\rangle$. The first aspect to notice is that for both logical states, the average photon number is $\bar{n} = 2$ and, as a result, the likelihood of a photon loss event is the same for both states. An observer detecting a loss event will therefore not gain any information allowing her to distinguish whether the loss came from $|0_L\rangle$ or from $|1_L\rangle$. This is a necessary condition for a quantum state encoded using the logical states Eq. 2.155 to not be 'deformed' by a photon loss event. Moreover, under the action of \hat{a} , the arbitrary superposition $c_0|0_L\rangle + c_1|1_L\rangle$ becomes $c_0|3\rangle + c_1|1\rangle$ after normalization. The coefficients c_0 and c_1 encoding the quantum information are intact and the original state can in principle be recovered with a unitary transformation. By noting that while the original state only has support on even photon numbers, the state after a photon loss only has support on odd photon numbers, we see that the photon loss event can be detected by measuring photon number parity $\hat{P} = (-1)^{\hat{n}}$. The parity operator thus plays the role of a stabilizer for this code.

This simple encoding should be compared to directly using the Fock states $|0\rangle, |1\rangle$ to store quantum information. Clearly, in this case, a single photon loss on $c_0|0\rangle + c_1|1\rangle$ leads to $|0\rangle$ and the quantum information has been irreversibly lost. Of course, this disadvantage should be contrasted to the fact that the rate at which photons are lost, which scales with \bar{n} , is (averaged over the code world) four times as large when using the encoding Eq. 2.155, compared to using the Fock states $|0\rangle, |1\rangle$. This observation reflects the usual conundrum of quantum error correction: using more resources (here more photons) to protect quantum information actually increases the natural error rate. The protocol for detecting and correcting errors must be fast enough and accurate enough to counteract this increase. The challenge for experimental implementations of quantum error correction is thus to reach and go beyond the break-even point where the encoded qubit, here Eq. 2.155, has a coherence time exceeding the coherence time of the unencoded constituent physical components, here the Fock states $|0\rangle, |1\rangle$. Near break-even performance with the above binomial code has been experimentally reported.

The simplest binomial code introduced above is able to correct a single amplitude-damping error (photon loss). Thus if the correction protocol is applied after a time interval δt , the probability of an uncorrectable error is reduced from $\mathcal{O}(\kappa\delta t)$ to $\mathcal{O}((\kappa\delta t)^2)$, where κ is the cavity energy decay rate.

To better understand the simplicity and efficiency advantages of bosonic QEC codes, it is instructive to do a head-to-head comparison of the simplest binomial code to the simplest qubit code for amplitude damping. The smallest qubit code able to protect logical information against a general single-qubit error requires five qubits. However, the specific case of the qubit amplitude damping channel can be corrected to first order against single-qubit errors using a 4-qubit code that, like the binomial code, satisfies the Knill-Laflamme conditions to

	4-qubit code	Simplest binomial code
Code word $ 0_L\rangle$	$\frac{1}{\sqrt{2}}(0000\rangle + 1111\rangle)$	$\frac{1}{\sqrt{2}}(0\rangle + 4\rangle)$
Code word $ 1_L\rangle$	$\frac{1}{\sqrt{2}}(1100\rangle + 0011\rangle)$	$ 2\rangle$
Mean excitation number n	2	2
Hilbert space dimension	$2^4 = 16$	$\{0, 1, 2, 3, 4\} = 5$
Number of correctable errors	$\{\hat{I}, \sigma_1^-, \sigma_2^-, \sigma_3^-, \sigma_4^-\} = 5$	$\{\hat{I}, a\} = 2$
Stabilizers	$\hat{S}_1 = \hat{Z}_1 Z_2, \hat{S}_2 = \hat{Z}_3 \hat{Z}_4, \hat{S}_3 = \hat{X}_1 \hat{X}_2 \hat{X}_3 \hat{X}_4$	$\hat{P} = (-1)^n$
Number of Stabilizers	3	1
Approximate QEC?	Yes, 1st order in γt	Yes, 1st order in κt

lowest order and whose two logical codewords are

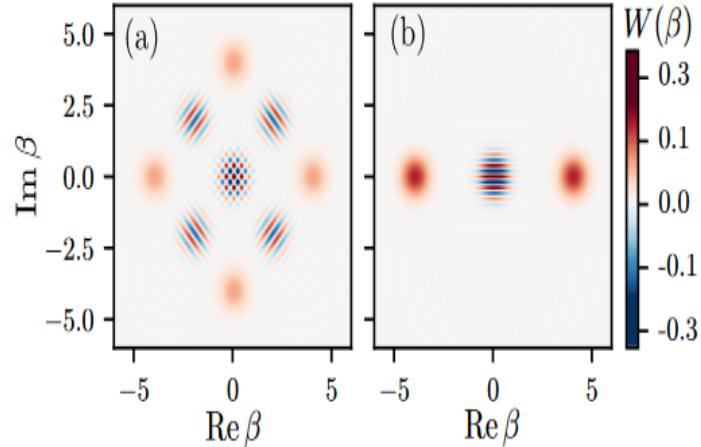
$$\begin{aligned} |0_L\rangle &= \frac{1}{\sqrt{2}}(|0000\rangle + |1111\rangle) \\ |1_L\rangle &= \frac{1}{\sqrt{2}}(|1100\rangle + |0011\rangle) \end{aligned} \quad (2.156)$$

This four-qubit amplitude damping code and the single-mode binomial bosonic code for amplitude damping are compared in Table 2.1. Note that, just as in the binomial code, both codewords have mean excitation number equal to two and so are equally likely to suffer an excitation loss. The logical qubit of Eq. 2.156 lives in a Hilbert space of dimension $2^4 = 16$ and has four different physical sites at which the damping error can occur. Counting the case of no errors, there are a total of five different error states which requires measurement of three distinct error syndromes $\hat{Z}_1 \hat{Z}_2$, $\hat{Z}_3 \hat{Z}_4$, and $\hat{X}_1 \hat{X}_2 \hat{X}_3 \hat{X}_4$ to diagnose (where \hat{P}_i refers to Pauli operator \hat{P} acting on qubit i). The required weight-two and weight-four operators have to date not been easy to measure in a highly QND manner and with high fidelity, but some progress has been made towards this goal. In contrast, the simple bosonic code in Eq. 2.155 requires only the lowest five states out of the (formally infinite) oscillator Hilbert space. Moreover, since there is only a single mode, there is only a single error, namely photon loss (or no loss), and it can be detected by measuring a single stabilizer, the photon number parity. It turns out that, unlike in ordinary quantum optics, photon number parity is relatively easy to measure in circuit QED with high fidelity and minimal state demolition. It is for all these reasons that, unlike the four-qubit code, the bosonic code Eq. 2.155 has already been demonstrated experimentally to (very nearly) reach the break-even point for QEC. Generalizations of this code to protect against more than a single photon loss event, as well as photon gain and dephasing are described.

Operation slightly exceeding break-even has been reported with cat-state bosonic encoding which we describe now. In the encoding used in that experiment, each logical code word is a superposition of four coherent states referred to as a four-component cat mode

$$\begin{aligned} |0_L\rangle &= \mathcal{N}_0(|\alpha\rangle + |i\alpha\rangle + |- \alpha\rangle + |- i\alpha\rangle) \\ |1_L\rangle &= \mathcal{N}_1(|\alpha\rangle - |i\alpha\rangle + |- \alpha\rangle - |- i\alpha\rangle) \end{aligned} \quad (2.157)$$

where \mathcal{N}_i are normalization constants, with $\mathcal{N}_0 \simeq \mathcal{N}_1$ for large $|\alpha|$. The Wigner functions for the $|0_L\rangle$ codeword is shown in Fig. 2.25 for $\alpha = 4$. The relationship between this encoding and the simple code in Eq. ?? can be seen by writing Eq. ?? using the expression Eq.2.84 for $|\alpha\rangle$ in terms of Fock states. One immediately finds that $|0_L\rangle$ only has support on Fock states $|4n\rangle$ with $n = 0, 1, \dots$, while $|1_L\rangle$ has support on Fock states $|4n + 2\rangle$, again for $n = 0, 1, \dots$. It follows that the two codewords are mapped onto orthogonal states under the action of \hat{a} , just as the binomial code of Eq.???. Moreover, the average photon number \bar{n} is approximately equal for the two logical states in the limit of large $|\alpha|$. The protection offered by this encoding is thus similar to that of the binomial code in Eq. 2.155. In fact, these two encodings belong to a larger class of codes characterized by rotation symmetries in phase space.



We end this section by discussing an encoding that is even simpler than Eq. 2.157, sometimes referred to as a two-component cat code. In this case, the codewords are defined simply as $|+_L\rangle = \mathcal{N}_0(|\alpha\rangle + |- \alpha\rangle)$ and $|-_L\rangle = \mathcal{N}_1(|\alpha\rangle - |- \alpha\rangle)$. The Wigner function for $|+_L\rangle$ is shown in Fig. 2.25(b). The choice to define the above codewords in the logical \hat{X}_L basis instead of the \hat{Z}_L basis is, of course, just a convention, but turns out to be convenient for this particular cat code. In contrast to Eqs. 2.155 and 2.157, these two states are *not* mapped to two orthogonal states under the action of \hat{a} . To understand this encoding, it is useful to consider the logical \hat{Z}_L basis states in the limit of large $|\alpha|$

$$\begin{aligned} |0_L\rangle &= \frac{1}{\sqrt{2}} (|+_L\rangle + |-_L\rangle) = |\alpha\rangle + \mathcal{O}\left(e^{-2|\alpha|^2}\right) \\ |1_L\rangle &= \frac{1}{\sqrt{2}} (|+_L\rangle - |-_L\rangle) = |-\alpha\rangle + \mathcal{O}\left(e^{-2|\alpha|^2}\right) \end{aligned} \quad (2.158)$$

As is made clear by the second equality, for large enough $|\alpha|$ these logical states are very close to coherent states of the same amplitude but opposite phase. The action of \hat{a} is thus, to a very good approximation, a phase flip since $\hat{a}|0_L/1_L\rangle \sim \pm|0_L/1_L\rangle$.

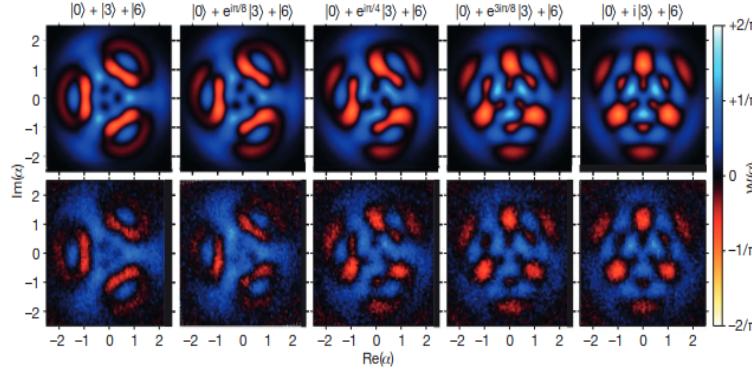
The advantage of this encoding is that, while photon loss leads to phase flips, the bit-flip rate is exponentially small with $|\alpha|$. This can be immediately understood from the golden rule whose relevant matrix element for bit flips is $\langle 1_L|\hat{a}|0_L\rangle \sim \langle -\alpha|\hat{a}|\alpha\rangle = \alpha e^{-2|\alpha|^2}$. In other words, if the qubit is encoded in a coherent state with many photons, losing one simply does not do much. This is akin to redundancy required for quantum error correction. As a result, the bit-flip rate ($1/T_1$) decreases *exponentially* with $|\alpha|^2$ while the phase flip rate increases only *linearly* with $|\alpha|^2$. The crucial point is that the bias between bit and phase flip error rates increases exponentially with α , which has been verified experimentally. While the logical states Eq. 2.158 do not allow for recovery from photon-loss errors, the strong asymmetry between different types of errors can be exploited to significantly reduce the qubit overhead necessary for fault-tolerant quantum computation. The basic intuition behind this statement is that the qubit defined by Eq. 2.158 can be used in an error correcting code tailored to predominantly correct the most likely error (here, phase flips) rather than devoting resources to correcting both amplitude and phase errors. Another bosonic encoding that was recently demonstrated in circuit QED is the Gottesman-Kitaev-Preskill (GKP) code. This demonstration is the first QEC experiment able to correct all logical errors and it came close to reaching the break-even point. While all the bosonic codes described above are based on codewords that obey rotation symmetry in phase space, the GKP code is instead based on translation symmetry. We will not describe the GKP encoding in more detail here.

2.2.9 QUANTUM OPTICS ON A CHIP

The strong light-matter interaction realized in circuit QED together with the flexibility allowed in designing and operating superconducting quantum circuits has opened the possibility to explore the rich physics of quantum optics at microwave frequencies in circuits. As discussed previously it has, for example, made possible the clear observation of vacuum Rabi splitting, of photon-number splitting in the strong-dispersive regime, as well as of signatures of ultrastrong light-matter coupling. The new parameter regimes that can be achieved in circuit QED have also made it possible to test some of the theoretical predictions from the early days of quantum optics and to explore new research avenues. A first indication that circuit QED is an ideal playground for these ideas is the strong Kerr nonlinearity relative to the decay rate, K/κ , that can readily be achieved in circuits. Indeed, from the point of view of quantum optics, a transmon is a Kerr nonlinear oscillator that is so nonlinear that it exhibits photon blockade. Given the very high Q factors that can be achieved in 3D superconducting cavities, such levels of nonlinearity can also readily be obtained in microwave resonators by using transmons or other Josephson junction-based circuits to induce

nonlinearity in electromagnetic modes.

Many of the links between circuit QED and quantum optics have already been highlighted. In this section, we continue this discussion by presenting some further examples.



2.2.9.1 Intra-cavity fields

Because superconducting qubits can rapidly be tuned over a wide frequency range, it is possible to bring them in and out of resonance with a cavity mode on a time scale which is fast with respect to $1/g$, the inverse of the qubit-cavity coupling strength. For all practical purposes, this is equivalent to the thought experiment of moving an atom in and out of the cavity in cavity QED. An experiment by Hofheinz took advantage of this possibility to prepare the cavity in Fock states up to $|n = 6\rangle$. With the qubit and the cavity in their respective ground states and the two systems largely detuned, their approach is to first π -pulse the qubit to its excited state. The qubit frequency is then suddenly brought in resonance with the cavity for a time $1/2g$ such as to swap the qubit excitation to a cavity photon as the system evolves under the Jaynes-Cummings Hamiltonian Eq. 2.22. The interaction is then effectively stopped by moving the qubits to its original frequency, after which the cycle is repeated until n excitations have been swapped in this way. Crucially, because the swap frequency between the states $|e, n-1\rangle$ and $|g, n\rangle$ is proportional to \sqrt{n} , the time during which qubit and cavity are kept in resonance must be adjusted accordingly at each cycle. The same \sqrt{n} dependence is then used to probe the cavity state using the qubit as a probe.

Building on this technique and using a protocol proposed by Law and Eberly for cavity QED, the same authors have demonstrated the preparation of arbitrary states of the cavity field and characterized these states by measuring the cavity Wigner function. Fig. 2.26 shows the result of this Wigner tomography for superpositions involving up to six cavity photons. A downside of this method is that the preparation time rapidly becomes comparable to the Fock state lifetime, limiting the Fock states which can be reached and the fidelity of the resulting states.

Taking the advantage of the very large χ/κ which can be reached in 3D cavities, an alternative to create such states is to cause qubit transitions conditioned on the Fock state of the cavity. Together with cavity displacements, these photon-number dependent qubit transitions can

be used to prepare arbitrary cavity states. Combining these ideas with numerical optimal control has allowed to synthesize cavity states with high fidelity such as Fock states up to $|n = 6\rangle$ and four-legged cat states.

The long photon lifetime that is possible in 3D superconducting cavities together with the possibility to realize a single-photon Kerr nonlinearity which overwhelms the cavity decay, $K/\kappa > 1$, has enabled a number of similar experiments such as the observation of collapse and revival of a coherent state in a Kerr medium and the preparation of cat states with nearly 30 photons. Another striking example is the experimental encoding of qubits in oscillator states already discussed in encoding a qubit in an oscillator section.

2.2.9.2 Quantum-limited amplification

Driven by the need for fast, high-fidelity single-shot readout of superconducting qubits, superconducting low-noise linear microwave amplifiers are a subject of intense research. There are two broad classes of linear amplifiers. First, phase-preserving amplifiers that amplify equivalently both quadratures of the signal. Quantum mechanics imposes that these amplifiers add a minimum of half a photon of noise to the input signal. Quantum mechanics imposes that these amplifiers add a minimum of half a photon of noise to the input signal. Second, phase-sensitive amplifiers which amplify one quadrature of the signal while squeezing the orthogonal quadrature. This type of amplifiers can in principle operate without adding noise. Amplifiers adding the minimum amount of noise allowed by quantum mechanics, phase preserving or not, are referred to as quantum-limited amplifiers. We note that, in practice, phase sensitive amplifiers are useful if the quadrature containing the relevant information is known in advance, a condition that is realized when trying to distinguish between two coherent states in the dispersive qubit readout discussed in dispersive qubit readout.

While much of the development of near-quantum limited amplifiers has been motivated by the need to improve qubit readout, Josephson junction based amplifiers have been theoretically investigated and experimentally demonstrated as early as the late 80's. These amplifiers have now found applications in a broad range of contexts. In their simplest form, such an amplifier is realized as a driven oscillator mode rendered weakly nonlinear by incorporating a Josephson junction and are generically known as a Josephson parametric amplifier (JPA).

For weak nonlinearity, the Hamiltonian of a driven nonlinear oscillator is well approximated by

$$H = \omega_0 \hat{a}^\dagger \hat{a} + \frac{K}{2} \hat{a}^{\dagger 2} \hat{a}^2 + \epsilon_p (\hat{a}^\dagger e^{-i\omega_p t} + \hat{a} e^{i\omega_p t}) \quad (2.159)$$

where ω_0 is the system frequency, K the negative Kerr nonlinearity, and ϵ_p and ω_p are the pump amplitude and frequency, respectively. The physics of the JPA is best revealed by applying a displacement transformation $\hat{D}^\dagger(\alpha) \hat{a} \hat{D}(\alpha) = a + \alpha$ to H with α chosen to cancel the pump term. Doing so leads to the transformed Hamiltonian

$$H_{\text{JPA}} = \delta \hat{a}^\dagger \hat{a} + \frac{1}{2} (\epsilon_2 \hat{a}^{\dagger 2} + \epsilon_2^* \hat{a}^2) + H_{\text{corr}} \quad (2.160)$$

where $\delta = \omega_0 + 2|\alpha|^2 K - \omega_p$ is the effective detuning, $\epsilon_2 = \alpha^2 K$, and are H_{corr} correction terms which can be dropped for weak enough pump amplitude and Kerr nonlinearity, when κ is large in comparison to K and thus the drive does not populate the mode enough for higher-order nonlinearity to become important. The second term, of amplitude ϵ_2 , is a two-photon pump which is the generator of quadrature squeezing. Depending on the size of the measurement bandwidth, this leads to phase preserving or sensitive amplification when operating the device close to but under the parametric threshold $\epsilon_2 < \sqrt{\delta^2 + (\kappa/2)^2}$, with κ the device's single photon loss rate. Rather than driving the nonlinear oscillator as in Eq. 2.159, an alternative approach to arrive at H_{JPA} is to replace the junction by a SQUID and to apply a flux modulation at $2\omega_0$.

Equation 2.160 is the Hamiltonian for a parametric amplifier working with a single physical oscillator mode. Using appropriate filtering in the frequency domain, single-mode parametric amplifiers can be operated in a phase-sensitive mode, when detecting the emitted radiation over the full bandwidth of the physical mode. This is also called the degenerate mode of operation. Alternatively, the same single-oscillator-mode amplifier can be operated in the phase-preserving mode, when separating frequency components above and below the pump in the experiment, e.g. by using appropriately chosen narrow-band filters. Parametric amplifiers with two or multiple physical modes are also frequently put to use the phase-sensitive and phase-preserving mode, e.g. in degenerate or non-degenerate mode of operation.

Important parameters which different approaches for implementing JPAs aim at optimizing include amplifier gain, bandwidth and dynamic range. The latter refers to the range of power over which the amplifier acts linearly, i.e. powers at which the amplifier output is linearly related to its input. Above a certain input power level, the correction terms in Eq. 2.160 resulting from the junction nonlinearity can no longer be ignored and lead to saturation of the gain. For this reason, while transmon qubits are operated in a regime where the single-photon Kerr nonlinearity is large and overwhelms the decay rate, JPAs are operated in a very different regime with $|K|/\kappa \sim 10^{-2}$ or smaller.

An approach to increase the dynamic range of JPAs is to replace the Josephson junction of energy E_J by an array of M junctions, each of energy ME_J . Because the voltage drop is now distributed over the array, the bias on any single junction is M times smaller and therefore the effective Kerr nonlinearity of the device is reduced from K to K/M^2 . As a result, nonlinear effects kick-in only at increased input signal powers leading to an increased dynamic range. Importantly, this can be done without degrading the amplifier's bandwidth. Typical values are ~ 50 MHz bandwidth with ~ -117 dBm saturation power for ~ 20 dB gain. Impedance engineering can be used to improve these numbers further.

Because the JPA is based on a localized oscillator mode, the product of its gain and bandwidth is approximately constant. Therefore, increase in one must be done at the expense of the other. As a result, it has proven difficult to design JPAs with enough bandwidth and dynamic range to simultaneously measure more than a few transmons.

To avoid the constant gain-bandwidth product which results from relying on a resonant mode, a drastically different strategy, known as the Josephson travelling-wave parametric amplifier (JTWP), is to use an open nonlinear medium in which the signal co-propagates

with the pump tone. While in a JPA the signal interacts with the nonlinearity for along time due to the finite Q of the circuit, in the JTWPAs the long interaction time is rather a result of the long propagation length of the signal through the nonlinear medium. In practice, JTWPAs are realized with a metamaterial transmission line whose center conductor is made from thousands of Josephson junctions in series. This device does not have a fixed gain-bandwidth product and has been demonstrated to have 20 dB over as much as 3 GHz bandwidth while operating close to the quantum limit. Because every junction in the array can be made only very weakly nonlinear, the JTWPAs also offers large enough dynamic range for rapid multiplexed simultaneously readout of multiple qubits.

2.2.9.3 Propagating fields and their characterization

1. Itinerant single and multi-photon states

In addition to using qubits to prepare and characterize quantum states of intra-cavity fields, it is also possible to take advantage of the strong nonlinearity provided by a qubit to prepare states of propagating fields at the output of a cavity. This can be done, for example, in a cavity with relatively large decay rate κ by tuning a qubit into and out of resonance with the cavity or by applying appropriately chosen drive fields. Alternatively, it is also possible to change the cavity decay rate in time to create single-photon states.

The first on-chip single-photon source in the microwave regime was realized with a dispersively coupled qubit engineered such that the Purcell decay rate γ_κ dominates the qubit's intrinsic non-radiative decay rate γ_1 . In this situation, exciting the qubit leads to rapid qubit decay by photon emission. In the absence of single-photon detectors working at microwave frequencies, the presence of a photon was observed by using a nonlinear circuit element (a diode) whose output signal is proportional to the square of the electric field, $\propto (\hat{a}^\dagger + \hat{a})^2$, and therefore indicative of the average photon number, $\langle \hat{a}^\dagger \hat{a} \rangle$, in repeated measurements.

Rather than relying on direct power measurements, techniques have also been developed to reconstruct arbitrary correlation functions of the cavity output field from the measurement records of the field quadratures. These approaches rely on multiple detection channels with uncorrelated noise to quantify and subtract from the data the noise introduced by the measurement chain. In this way, it is possible to extract, for example, first- and second-order coherence functions of the microwave field. Remarkably, with enough averaging, this approach does not require quantum-limited amplifiers, although the number of required measurement runs is drastically reduced when such amplifiers are used when compared to HEMT amplifiers.

This approach was used to measure second-order coherence functions, $G^2(t, t + \tau) = \langle \hat{a}^\dagger(t) \hat{a}^\dagger(t + \tau) \hat{a}(t + \tau) \hat{a}(t) \rangle$, in the first demonstration of antibunching of a pulsed single microwave-frequency photon source. The same technique also enabled the observation of resonant photon blockade at microwave frequencies and, using two single photon sources at the input of a microwave beam splitter, the indistinguishability of microwave photon was demonstrated in a Hong-Ou-Mandl correlation function measurement. Moreover, a similar approach was used to characterize the blackbody radiation emitted by a 50 Ω load resistor.

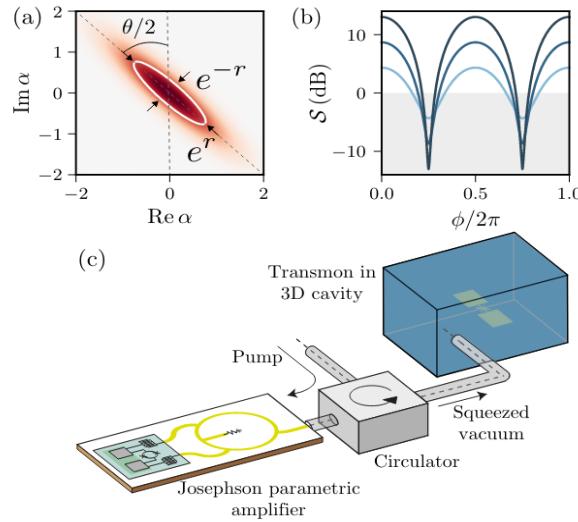
Building on these results, it is also possible to reconstruct the quantum state of itinerant microwave fields from measurement of the fields moments. This technique relies on interleaving two types of measurements: measurements on the state of interest and ones in which the field is left in the vacuum as a reference to subtract away the measurement chain noise. In this way, the Wigner function of arbitrary superpositions of vacuum and one-photon Fock states have been reconstructed. This technique was extended to propagating modes containing multiple photons. Similarly, entanglement between a (stationary) qubit and a propagating mode was quantified in this approach with joint state tomography. Quadrature-histogram analysis also enabled, for example, the measurement of correlations between radiations fields, and the observation of entanglement of itinerant photon pairs in waveguide QED.

2. Squeezed microwave fields

Operated in the phase-sensitive mode, quantum-limited amplifiers are sources of squeezed radiation. Indeed, for $\delta = 0$ and ignoring the correction terms, the JPA Hamiltonian of Eq. 2.160 is the generator of the squeezing transformation

$$S(\zeta) = e^{\frac{1}{2}\zeta^*\hat{a}^2 - \frac{1}{2}\zeta\hat{a}^\dagger 2} \quad (2.161)$$

which takes vacuum to squeezed vacuum, $|\zeta\rangle = S(\zeta)|0\rangle$. In this expression, $\zeta = re^{i\theta}$ with r the squeezing parameter and θ the squeezing angle. As illustrated in Fig. 2.27(a), the action of $S(\zeta)$ on vacuum is to 'squeeze' one quadrature of the field at the expense of 'anti-squeezing' the orthogonal quadrature while leaving the total area in phase space unchanged. As a result, squeezed states, like coherent states, saturate the Heisenberg inequality.



This can be seen more clearly from the variance of the quadrature operator \hat{X}_ϕ which takes the form

$$\Delta X_\phi^2 = \frac{1}{2} \left(e^{2r} \sin^2 \tilde{\phi} + e^{-2r} \cos^2 \tilde{\phi} \right) \quad (2.162)$$

where we have defined $\tilde{\phi} = \phi - \theta/2$. In experiments, the squeezing level is often reported in dB computed using the expression

$$S = 10 \log_{10} \frac{\Delta X_\phi^2}{\Delta X_{vac}^2} \quad (2.163)$$

Figure 2.27(b) shows this quantity as a function of ϕ . It reaches its minimal value $e^{-2r}/2$ at $\phi = [\theta + (2n + 1)\pi]/2$ where the variance ΔX_ϕ^2 dips below the vacuum noise level $\Delta X_{vac}^2 = 1/2$ (horizontal line).

Squeezing in Josephson devices was observed already in the late 80's, experiments that have been revisited with the development of near quantum-limited amplifiers. Quantum state tomography of an itinerant squeezed state at the output of a JPA was reported. There, homodyne detection with different LO phases on multiple preparations of the same squeezed state, together with maximum likelihood techniques, was used to reconstruct the Wigner function of the propagating microwave field. Moreover, the photon number distribution of a squeezed field was measured using a qubit in the strong dispersive regime. As is clear from the form of the squeezing transformation $S(\zeta)$, squeezed vacuum is composed of a superposition of only even photon numbers, something which confirmed in experiments.

Thanks to the new parameter regimes that can be achieved in circuit QED, it is possible to experimentally test some long-standing theoretical predictions of quantum optics involving squeezed radiation. For example, in the mid-80's theorists predicted how dephasing and resonance fluorescence of an atom would be modified in the presence of squeezed radiation. Experimentally testing these ideas in the context of traditional quantum optics with atomic systems, however, represents a formidable challenge. The situation is different in circuits where squeezed radiation can easily be guided from the source of squeezing to the qubit playing the role of artificial atom. Moreover, the reduced dimensionality in circuits compared to free-space atomic experiments limits the number of modes that are involved, such that the artificial atom can be engineered so as to preferentially interact with a squeezed radiation field.

Taking advantage of the possibilities offered by circuit QED, confirmed the prediction that squeezed radiation can inhibit phase decay of an (artificial) atom. In this experiment, the role of the two-level atom was played by the hybridized cavity-qubit state $\{|g,0\rangle, |e,0\rangle\}$. Moreover, squeezing was produced by a JPA over a bandwidth much larger than the natural linewidth of the two-level system, see Fig. 2.27(c). According to theory, quantum noise below the vacuum level along the squeezed quadrature leads to a reduction of dephasing. Conversely, along the anti-squeezed quadrature, the enhanced fluctuations lead to increased dephasing along orthogonal axis of the Bloch sphere. In the experiment, phase decay inhibition along the squeezed quadrature was such that the associated dephasing time increased beyond the usual vacuum limit of $2T_1$. By measuring the dynamics of the two-level atom, it was moreover possible to reconstruct the Wigner distribution of the itinerant squeezed state produced by the JPA. Using a similar setup, resonance fluorescence in the presence of squeezed vacuum and found excellent agreement with theoretical predictions studied. In this

way, it was possible to infer the level of squeezing (3.1 dB below vacuum) at the input of the cavity.

The discussion has so far been limited to squeezing of a single mode. It is also possible to squeeze a pair of modes, which is often referred to as two-mode squeezing. Labeling the modes as \hat{a}_1 and \hat{a}_2 , the corresponding squeezing transformation reads

$$S_{12}(\zeta) = e^{\frac{1}{2}} \zeta^* \hat{a}_1 \hat{a}_2 - \frac{1}{2} \zeta \hat{a}_1^\dagger \hat{a}_2^\dagger \quad (2.164)$$

Acting on vacuum, S_{12} generates a two-mode squeezed state which is an entangled state of modes \hat{a}_1 and \hat{a}_2 . As a result, in isolation, the state of one of the two entangled modes appears to be in a thermal state where the role of the Boltzmann factor $\exp(-\beta \hbar \omega_i)$, with $\omega_i =_{1,2}$ the mode frequency, is played by $\square \dashv h^2 r$. In this case, correlations and therefore squeezing is revealed when considering joint quadratures of the form $\hat{X}_1 \pm \hat{X}_2$ and $\hat{P}_1 \pm \hat{P}_2$, rather than the quadratures of a single mode as in Fig. 2.27(a). In Josephson-based devices, two-mode squeezing can be produced using mode nondegenerate parametric amplifiers of different frequencies, often referred to as signal and idler in this context. Other experiments have demonstrated two-mode squeezing in two different spatial modes, i.e. entangled signals propagating along different transmission lines.

2.2.9.4 Remote Entanglement Generation

Several approaches to entangle nearby qubits have been discussed in quantum computing with circuit QED section. Together with protocols such as quantum teleportation, entanglement between distant quantum nodes can be the basis of a ‘quantum internet’. Because optical photons can travel for relatively long distances in room temperature optical fiber while preserving their quantum coherence, this vision appears easier to realize at optical than at microwave frequencies. Nevertheless, given that superconducting cables at millikelvin temperatures have similar losses per meter as optical fibers, there is no reason to believe that complex networks of superconductor-based quantum nodes cannot be realized. One application of this type of network is a modular quantum computer architecture where the nodes are relatively small-scale error-corrected quantum computers connected by quantum links.

One approach to entangle qubits fabricated in distant cavities relies on entanglement by measurement, which is easy to understand in the case of two qubits coupled to the same cavity. Assuming the qubits coupled to the same cavity. Assuming the qubits to have the same dispersive shift χ due to coupling to the cavity, the dispersive Hamiltonian in a doubly rotating frame takes the form

$$H = \chi (\hat{\sigma}_{z1} + \hat{\sigma}_{z2}) \hat{a}^\dagger \hat{a} \quad (2.165)$$

Crucially, the cavity pull associated with odd-parity states $\{|01\rangle, |10\rangle\}$ is zero while it is $\pm 2\chi$ for the even-parity states $\{|00\rangle, |11\rangle\}$. As a result, for $\chi \gg \kappa$, a tone at the bare cavity frequency leads to a large cavity field displacement for the even-parity subspace. On the other hand, the displacement is small or negligible for the odd-parity subspace. Starting with a

uniform unentangled superposition of the states of the qubits, homodyne detection therefore stochastically collapses the system of these subspaces thereby preparing an entangled state of the two qubits, an idea that was realized experimentally.

The same concept was used in 2014 to entangle two transmon qubits coupled to two 3D cavities separated by more than a meter of coaxial cable. There, the measurement tone transmitted through the first cavity, only after which it is measured by homodyne detection therefore stochastically collapses the system to one of these subspaces thereby preparing an entangled state of the two qubits, an idea that was realized experimentally.

The same concept was used to entangle two transmon qubits coupled to two 3D cavities separated by more than a meter of coaxial cable. There, the measurement tone transmitted through the first cavity is sent to the second cavity, only after which it is measured by homodyne detection. In this experiment, losses between the two cavities -mainly due to the presence of a circulator preventing any reflection from the second cavity back to the first cavity- as well as finite detection efficiency as the main limit to the achievable concurrence, a measure of entanglement, to 0.35.

While the above protocol probabilistically entangles a pair of qubits, a more powerful but also more experimentally challenging approach allows, in principle, to realize this in a fully deterministic fashion. Developed in the context of cavity QED, this scheme relies on mapping the state of an atom strongly coupled to a cavity to a propagating photon. By choosing its wave packet to be time-symmetric, the photon is absorbed with unit probability by a second cavity also containing an atom. In this way, it is possible to exchange a quantum state between the two cavities. Importantly, this protocol relies on having a unidirectional channel between the cavities such that no signal can propagate from the second to the first cavity. At microwave frequencies, this is achieved by inserting a circulator between the cavities. By first entangling the emitter qubit to a partner qubit located in the same cavity, the quantum-state transfer protocol can be used to entangle the two nodes.

Variations on this more direct approach to entangle remote nodes have been implemented in circuit QED. All three experiments rely on producing time-symmetric propagating photons by using the interaction between a transmon qubit and cavity mode. Multiple approaches to shape and catch propagating photons have been developed in circuit QED. For example, a transmission-line resonator with a tunable input port to catch a shaped microwave pulse with over 99% probability is used. Time-reversal-symmetric photon have been created using 3-island transmon qubits in which the coupling to a microwave resonator is controlled in time so as to shape the mode function of spontaneously emitted photons. In a similar fashion, shaped single photons can be generated by modulating the boundary condition of a semi-infinite transmission line using a SQUID which effectively controls the spontaneous emission rate of a qubit coupled to the line and emitting the photon.

Alternatively, the remote entanglement generation experiment rather relies on a microwave-induced amplitude and phase-tunable coupling between the qubit-resonator $|f0\rangle$ and $|g1\rangle$ states, akin to the fg-ge gate already mentioned in all-microwave gates. Exciting the qubit to its $|f\rangle$ state followed by a π -pulse on the $f0-g1$ transition transfers the qubit excitation to a single resonator photon which is emitted as a propagating photon. This single-photon wave

packet can be shaped to be time-symmetric by tailoring the envelope of the $f_0 - g_1$ pulse. By inducing the reverse process with a time-reversed pulse on a second resonator photon which is emitted as a propagating photon. This single-photon wave packet can be shaped to be time-symmetric by tailoring the envelope of the $f_0 - g_1$ pulse. By inducing the reverse process with a time reversed pulse on a second resonator also containing a transmon, the itinerant photon is absorbed by this second transmon. In this way, an arbitrary quantum state can be transferred with a probability of 98.1 % between the two cavities separated by 0.9 m of coaxial line bisected by a circulator. By rather preparing the emitter qubit in a $(|e\rangle + |f\rangle)/\sqrt{2}$ superposition, the same protocol deterministically prepares an entangled state of the two transmons with a fidelity of 78.9 % at a rate of 50 kHz. The experiments reported similar Bell-state fidelities using different approaches to prepare time-symmetric propagating photons. The fidelity reported by the three experiments suffered from the presence of a circulator bisecting the nearly one meter-long coaxial cable separating the two nodes. Replacing the lossy commercial circulator by an on-chip quantum-limited version could improve the fidelity. By taking advantage of the multimode nature of a meter long transmission line, it was also possible to deterministically entangle remote qubits without the need of a circulator. In this way, a bidirectional communication channel between the nodes is established and deterministic Bell pair production with 79.3% fidelity has been reported.

2.2.9.5 Waveguide QED

The bulk of this review is concerned with the strong coupling of artificial atoms to the confined electromagnetic field of a cavity. Strong light-matter coupling is also possible in free space with an atom or large dipole-moment molecule by tightly confining an optical field in the vicinity of the atom or molecule. A signature of strong coupling in this settings is the extinction of the transmitted light by the single atom or molecule acting as a scatterer. This extinction results from destructive interference of the light beam with the collinearly emitted radiation from the scatterer. Ideally, this results in 100% reflection. In practice, because the scatterer emits in all directions, there is poor mode matching with the focused beam and reflection of $\sim 10\%$ is observed with a single atom and $\sim 30\%$ with a single molecule.

Mode matching can, however, be made to be close to ideal with electromagnetic fields in 1D superconducting transmission lines and superconducting artificial atoms where the artificial atoms can be engineered to essentially only emit in the forward and backward directions along the line. In the first realization of this idea in superconducting quantum circuits, extinction of the transmitted signal by much as 94% by coupling a single flux qubit to a superconducting transmission line is observed. Experiments with a transmon qubit have seen extinction as large as 99.6%. Pure dephasing and non-radiative decay into other modes than the transmission line are the cause of the small departure from ideal behavior in these experiments. Nevertheless, the large observed extinction is a clear signature that radiative decay in the transmission line γ_{nr} (i.e. Purcell decay) overwhelms non-radiative decay γ_{nr} . In short, in this cavity-free system referred to as waveguide QED, $\gamma_r/\gamma_{nr} \gg 1$ is the appropriate definition of strong coupling and is associated with a clear experimental signature: the extinction of transmission by a single scatterer.

Despite its apparent simplicity, waveguide QED is a rich toolbox with which a number of

physical phenomena have been investigated. This includes Autler-Townes splitting, single-photon routing, the generation of propagating nonclassical microwaves states, as well as large cross-Kerr phase shifts at the single-photon level.

In another experiment, the radiative decay of an artificial atom is studied which is placed in front of a mirror, here formed by a short to ground of the waveguide's center conductor. In the presence of a weak drive field applied to the waveguide, the atom relaxes by emitting a photon in both directions of the waveguide. The radiation emitted towards the mirror, assumed here to be on the left of the atom, is reflected back to interact again with the atom after having acquired a phase shift $\theta = 2 \times 2\pi l/\lambda + \pi$, where l is the atom-mirror distance and λ the wavelength of the emitted radiation. The additional phase factor of π accounts for the hard reflection at the mirror. Taking into account the resulting multiple round trips, this modifies the atomic radiative decay rate which takes the form $\gamma(\theta) = 2 * \text{gamma}_r \rfloor \{f^2(\theta/2)$.

For $l/\lambda = 1/2$, the radiative decay rate vanishes corresponding to destructive interference of the right-moving field and the left-moving field after multiple reflections on the mirror. In contrast, for $l/\lambda = 1/4$, these fields interfere constructively leading to enhanced radiative relaxation with $\gamma(\theta) = 2\gamma_r$. The ratio l/λ can be modified by shorting the waveguide's center conductor with a SQUID. In this case, the flux threading the SQUID can be used to change the boundary condition seen by the qubit, effectively changing the distance l . The experiment of Hoi rather relied on flux-tuning of the qubit transition frequency, thereby changing λ . In this way, a modulation of the qubit decay rate by a factor close to 10 was observed. A similar experiment has been reported with a trapped ion in front of a movable mirror.

Engineering vacuum fluctuations in this system has been pushed even further by creating microwave photonic bandgaps in waveguides to which transmon qubits are coupled. For example, another experiment has coupled a transmon qubit to a metamaterial formed by periodically loading the waveguide with lumped-element microwave resonators. By tuning the transmon frequency in the band gap where there is zero or only little density of states to accept photons emitted by the qubit, an increase by a factor of 24 of the qubit lifetime was observed.

An interpretation of the 'atom in front of a mirror' experiments is that the atom interacts with its mirror image. Rather than using a boundary condition (i.e. a mirror) to study the resulting constructive and destructive interferences and change in the radiative decay rate, it is also possible to couple a second atom to the same waveguide. In this case, photons emitted by one atom can be absorbed by the second atom leading to interactions between the atoms separated by a distance $2l$. Similar to the case of a single atom in front of a mirror, when the separation between the atoms is such that $2l/\lambda = 1/2$, correlated decay of the pair of atoms at the enhanced rate $2\gamma_1$ is expected and experimentally observed. On the other hand, at a separation of $2l/\lambda = 3/4$, correlated decay is replaced by coherent energy exchange between the two atoms mediated by virtual photons. We note that the experiments of van Loo with transmon qubits agree with the waveguide. Derivations from these predictions are expected as the distance between the atoms increases.

Finally, following a proposal of an experiment used a pair of transmon qubits to act as an

effective cavity for a third transmon qubit, all qubits being coupled to the same waveguide. In this way, vacuum Rabi oscillations between the dark state of the effective cavity and the qubit playing the role of atom were observed, confirming that the strong-coupling regime of cavity QED was achieved.

2.2.9.6 Single microwave photon detection

The development of single-photon detectors at infrared, optical and ultraviolet frequencies has been crucial to the field of quantum optics and in fundamental test of quantum physics. High-efficiency photon detectors are, for example, one of the elements that allowed the loophole-free violation of Bell's inequality. Because microwave photons have orders of magnitude less energy than infrared, optical or ultraviolet photons, the realization of a photon detector at microwave frequencies is more challenging. Yet, photon detectors in that frequency range would find a number of applications, including in quantum information processing, for quantum radars and for the detection of dark matter axions.

Non-destructive counting of microwave photons localized in a cavity has already been demonstrated experimentally by using an (artificial) atom as a probe in the strong dispersive regime. Similar measurements have also been done using a transmon qubit mediating interactions between two cavities, one containing the photons to be measured and a second acting as a probe. The detection of itinerant microwave photons remains, however, more challenging. A number of theoretical proposals have appeared. One common challenge for these approaches based on absorbing itinerant photons in a localized mode before detecting them can be linked to the quantum Zeno effect. Indeed, continuous monitoring of the probe mode will prevent the photon from being absorbed in the first place. Approaches to mitigate this problem have been introduced, including using an engineered, impedance matched Λ -system used to deterministically capture the incoming photon, and using the bright and dark-states of an ensemble of absorbers.

Despite these challenges, first itinerant microwave photon detectors have been achieved in the laboratory, in some cases achieving photon detection without destroying the photon in the process. Notably, a microwave photon counter was used to measure a superconducting qubit with a fidelity of 92% without using a linear amplifier between the source and the detector. Despite these advances, the realization of a high-efficiency, large-bandwidth, QND single microwave photon detector remains a challenge for the field.

2.2.10 OUTLOOK

Fifteen years after its introduction, circuit QED is a leading architecture for quantum computing and an exceptional platform to explore the rich physics of quantum optics in new parameter regimes. Circuit QED has, moreover, found applications in numerous other fields of research as discussed in the body of the review and in the following. In closing this review, we turn to some of these recent developments.

Although there remain formidable challenges before large-scale quantum computation becomes a reality, the increasing number of qubits that can be wired up, as well as the

improvements in coherence time and gate fidelity, suggests that it will eventually be possible to perform computations on circuit QED-based quantum processors that are out of reach of current classical computers. Quantum supremacy on a 53-qubit device has already been claimed, albeit on a problem of no immediate practical interest. There is, however, much effort deployed in finding useful computational tasks which can be performed on Noisy Intermediate-Scale Quantum (NISQ) devices. First step in this direction include the determination of molecular energies with variational quantum eigensolvers or boson sampling approaches and machine learning with quantum-enhanced features.

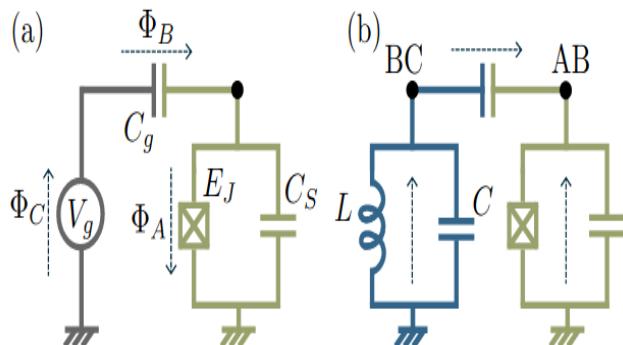
Engineered circuit QED-based devices also present an exciting avenue toward performing analog quantum simulations. In contrast to quantum computing architectures, quantum simulators are usually tailored to explore a single specific problem. An example are arrays of resonators which are capacitively coupled to allow photons to hop from resonator to resonator. Taking advantage of the flexibility of superconducting quantum circuits, it is possible to create exotic networks of resonators such as lattices in an effective hyperbolic space with constant negative curvature. Coupling qubits to each resonator realizes a Jaynes-Cummings lattice which exhibits a quantum phase transition similar to the superfluid-Mott insulator transition in BoseHubbard lattices. Moreover, the nonlinearity provided by capacitively coupled qubits, or Josephson junctions embedded in the center conductor of the resonators, creates photon-photon interactions. This leads to effects such as photon blockade bearing some similarities to Coulomb blockade in mesoscopic systems. Few resonator- and qubit-devices are also promising for analog quantum simulations. Examples are the exploration of a simple model of the light harvesting process in photosynthetic complexes in a circuit QED device under the influence of both coherent and incoherent drives, and the analog simulation of dissipatively stabilized strongly correlated quantum matter in a small photon BoseHubbard lattice.

Because it is a versatile platform to interface quantum devices with transition frequencies in the microwave domain to photons stored in superconducting resonators at similar frequencies, the ideas of circuit QED are also now used to couple to a wide variety of physical systems. An example of such hybrid quantum systems are semiconducting microwave resonators. Here, the position of an electron in a double dot leads to a dipole moment to which the resonator electric field couples. First experiments with gate-defined double quantum dots in nanotubes, and InAs nanowires have demonstrated dispersive coupling and its use for characterizing charge states of quantum dots. These first experiments were, however, limited by the very large dephasing rate of the quantum dot's charge states, but subsequent experiments have been able to reach the strong coupling regime. Building on these results and by engineering an effective spin-orbit interaction, it has been possible to reach the strong coupling regime with single spins.

When the coupling to a single spin cannot be made large enough to reach the strong coupling regime, it can be possible to rely on an ensemble of spins to boost the effective coupling. Indeed, in the presence of an ensemble of N emitters, the coupling strength to the ensemble is enhanced by \sqrt{N} , such that for large enough $g\sqrt{N}$ the strong coupling regime can be reached. First realization of these ideas used ensembles of $\sim 10^{12}$ spins to bring the coupling from a few Hz to ~ 10 MHz with NV centers in diamond and Cr^{3+} spins in ruby. One objective

of these explorations is to increase the sensitivity of electron paramagnetic resonance (EPR) or electron spin resonance (ESR) spectroscopy for spin detection with the ultimate goal of achieving the single-spin limit. A challenge in reaching this goal is the long lifetime of single spins in these systems which limits the repetition rate of the experiment. By engineering the coupling between the spins and an LC oscillator fabricated in close proximity, it has been possible to take advantage of the Purecell effect to reduce the relaxation time from 10^3 s to 1s. This faster time scale allows for faster repetition rates thereby boosting the sensitivity, which could lead to spin sensitivities on the order of $0.1 \text{ spin}/\sqrt{\text{Hz}}$.

Mechanical systems operated in the quantum regime also benefited from the ideas of circuit QED. An example is a suspended aluminium membrane that plays the role of a vacuum gap capacitor in a microwave LC oscillator. The frequency of this oscillator depends on the separation between the plates of the capacitor leading to a coupling between the oscillator and the flexural mode of the membrane. Strong coupling between mechanical motion and the LC oscillator has been demonstrated, which allowed to sideband cool the motion of the mechanical oscillator to phonon occupation number as small as $n_{\text{phonon}} \sim 0.34$. Squeezed radiation generated by a Josephson parametric amplifier was also used to cool beyond the quantum backaction limit to $n_{\text{phonon}} \sim 0.19$. Building on these ideas, entanglement of the mechanical motion and the microwave fields was demonstrated as well as coherent state transfer between itinerant microwave fields and mechanical oscillator.



Hybrid systems are also important in the context of microwave to optical frequency transduction in the quantum regime. This is a very desirable primitive for quantum networks, as it would allow quantum processors based on circuit QED to be linked optically over large distances. A variety of hybrid systems are currently being investigated for this purpose, including electro-optomechanical, electro-optic and magneto-optics ones. Two other hybrid quantum systems that have recently emerged are quantum surface acoustic waves interacting with superconducting qubits, and quantum magnonics where quanta of excitation of spin-wave modes known as magnon are strongly coupled to the field of a 3D microwave cavity.

Chapter 3

Quantum Processor Design

This section contains hands on tutorials from IBM that uses Qiskit Metal for quantum processor design.

3.1 Overview

3.1.0.0.1 You'll use Qiskit Metal in 4 stages

1. Choose a design class to instantiate.
2. Add and modify pre-built components (qubits, coplanar wave guides, etc.) from the QComponent library to your design.
3. Render to Simulate & Analyze
 - Current Rendering Options:
 - Ansys
 - * HFSS Renderer - for high frequency simulations (eigenmode, modal, terminal)
 - EPR Analysis - Uses eigenmode simulation to perform energy participation ratio analysis
 - * Q3D Renderer - for extracting equivalent circuit values of a layout, such as capacitance
 - LOM Analysis - Uses the capacitance matrix from Q3D to determine the parameters of a transmon qubit
4. Render for Fabrication
 - Current Rendering Options:
 - GDS

These steps are shown visually below in the following diagram

3.1.0.1 This tutorial is for steps 1 and 2.

3.1.0.2 Using this Tutorial

Metal can be used three different ways:

- * Jupyter Notebooks
- * For interactive code
- * To Use: 1. Just press run :D
- * Python scripts
- * For setting internal breakpoints
- * To Use: 1. Copy snippets of code from these Notebooks and save as a Python file.
- 2. Run in your favorite editor. (We like VS Code!)
- * Metal GUI
- * *In the future, we anticipate Metal GUI to have full functionality.*
- * To Use: 1. You *must* first use either Jupyter Notebooks or Python Scripts to add components to your QDesign.
- 2. Use the GUI to visualize and manually edit your components.

Let's dive in!

3.1.0.3 QDesign (need-to-know)

Each time you create a new quantum circuit design, you start by instantiating a QDesign class.

There are different design classes in the design library `qiskit_metal.designs` for different design layouts. For example the design class `DesignPlanar` is best for 2D circuit designs.

Every design class (except `QDesign`) inherits from the base `QDesign` class. `QDesign` defines basic functionality for all other design classes and should not be directly instantiated.

3.1.0.4 `QDesign` (in-depth)

`QDesign` keeps track of each of the components (qubits, coplanar wave guides, etc.) that you add to your circuit and the relationships between them.

- `QComponents` - do *not* directly instantiate
 - Components of your design
 - Example:
 - * Transmon Qubits
 - * CPWs
 - * etc.
 - Upon creation, the `QComponent`'s `make` function runs and adds the `QComponent`'s geometries (rectangles, line segments, etc.) to the `QGeometryTables`
- `QGeometryTables` - instantiate during init of `QDesign`
 - Stores backend information about components
 - Populated when `QComponents` are added to `QDesign`
- `QNet.net_info` - instantiate during init of `QDesign`
 - Stores backend information on existing connections between components
 - Instantiated in the backend by `QDesign`
 - Populated during connections of `QComponents`
- `QRenderer` - instantiate during init of `QDesign`
 - This is what allows you to export your designs into Ansys, GDS, etc.
 - `qiskit_metal/config.py` contains list of all instantiated renderers

3.2 Coding Time!

Today we'll be creating a 2D design and adding a single qcomponent

So, let us dive right in. For convenience, let's begin by enabling [automatic reloading of modules](#) when they change.

```
[1]: %load_ext autoreload  
%autoreload 2
```

3.2.1 Import Qiskit Metal

```
[2]: import qiskit_metal as metal  
from qiskit_metal import designs, draw  
from qiskit_metal import MetalGUI, Dict, open_docs  
  
%metal_heading Welcome to Qiskit Metal!
```

```
<IPython.core.display.HTML object>
```

Here, we import the folders designs, draw, MetalGUI, Dict, and open_docs from the qiskit_metal code.

3.2.2 My First Quantum Design (QDesign)

Choose a design layout. We will start with the simple planar QDesign.

```
[3]: design = designs.DesignPlanar()
```

```
[4]: # Since we are likely to be making many changes while tuning and
      → modifying our design, we will enable overwriting.
      # If you disable the next line, then you will need to delete a component
      → [<component>.delete()] before recreating it.
```

```
design.overwrite_enabled = True
```

```
[5]: %metal_heading Hello Quantum World!
```

```
<IPython.core.display.HTML object>
```

```
[6]: # We can also check all of the chip properties to see if we want to
      → change the size or any other parameter.
      # By default the name of chip is "main".
      design.chips.main
```

```
[6]: {'material': 'silicon',
      'layer_start': '0',
      'layer_end': '2048',
      'size': {'center_x': '0.0mm',
              'center_y': '0.0mm',
              'center_z': '0.0mm',
              'size_x': '9mm',
              'size_y': '6mm',
              'size_z': '-750um',
              'sample_holder_top': '890um',
              'sample_holder_bottom': '1650um'}}}
```

```
[7]: design.chips.main.size.size_x = '11mm'
      design.chips.main.size.size_y = '9mm'
```

Launch Qiskit Metal GUI to interactively view, edit, and simulate QDesign: Metal GUI

```
[8]: gui = MetalGUI(design)
```

3.2.3 My First Quantum Component (QComponent)

3.2.3.1 A transmon qubit

You can create a ready-made transmon qubit from the QComponent Library, `qiskit_metal qlibrary.qubits`. `transmon_pocket.py` is the file containing our qubit so `transmon_pocket` is the module we import. The `TransmonPocket` class is our transmon qubit. Like all quantum components, `TransmonPocket` inherits from `QComponent`

- Let's create a new qubit by creating an object of this class.

```
[9]: # Select a QComponent to create (The QComponent is a python class named
      ↪ `TransmonPocket`)
from qiskit_metal qlibrary.qubits.transmon_pocket import TransmonPocket

q1 = TransmonPocket(design, 'Q1',
                     ↪ options=dict(connection_pads=dict(a=dict())))
# Create a new Transmon
# ↪ Pocket object with name 'Q1'
gui.rebuild() # rebuild the design and plot

gui.edit_component('Q1') # set Q1 as the editable component
gui.autoscale() # resize GUI to see QComponent
```

Let's see what the `Q1` object looks like

```
[10]: q1 #print Q1 information
```

```
[10]: name:    Q1
      class:   TransmonPocket
      options:
      'pos_x'           : '0.0um',
      'pos_y'           : '0.0um',
      'orientation'     : '0.0',
      'chip'            : 'main',
      'layer'           : '1',
      'connection_pads': {
          'a'             : {
              'pad_gap'       : '15um',
              'pad_width'     : '125um',
              'pad_height'    : '30um',
              'pad_cpw_shift' : '5um',
              'pad_cpw_extent': '25um',
              'cpw_width'     : 'cpw_width',
              'cpw_gap'       : 'cpw_gap',
              'cpw_extend'    : '100um',
```

```

'pocket_extent'      : '5um',
'pocket_rise'        : '65um',
'loc_W'              : '+1',
'loc_H'              : '+1',
    },
},
'pad_gap'            : '30um',
'inductor_width'    : '20um',
'pad_width'          : '455um',
'pad_height'         : '90um',
'pocket_width'       : '650um',
'pocket_height'      : '650um',
'hfss_wire_bonds'   : False,
'q3d_wire_bonds'    : False,
'hfss_inductance'   : '10nH',
'hfss_capacitance'  : 0,
'hfss_resistance'   : 0,
'hfss_mesh_kw_jj'    : 7e-06,
'q3d_inductance'    : '10nH',
'q3d_capacitance'   : 0,
'q3d_resistance'    : 0,
'q3d_mesh_kw_jj'     : 7e-06,
'gds_cell_name'      : 'my_other_junction',
module: qiskit_metal qlibrary qubits transmon_pocket
id: 1

```

3.2.3.1.1 What are the default options?

The QComponent comes with some default options like the length of the pads for our transmon pocket.

- * Options are parsed internally by Qiskit Metal via the component's `make` function.
- * You can change option parameters from the gui or the script api.

[11]: %metal_print How do I edit options? API or GUI

```
<IPython.core.display.HTML object>
```

You can now use the Metal GUI to edit, plot, and modify quantum components. Equivalently, you can also do everything from the Jupyter Notebooks/Python scripts (which call the Python API directly). The GUI is just calling the Python API for you.

*You must use a string when setting options!

[12]: # Change options

```

q1.options.pos_x = '0.5 mm'
q1.options.pos_y = '0.25 mm'
q1.options.pad_height = '225 um'

```

```
q1.options.pad_width = '250 um'  
q1.options.pad_gap = '50 um'
```

```
[13]: gui.rebuild() # Update the component geometry, since we changed the  
       ↪options  
  
# Get a list of all the qcomponents in QDesign and then zoom on them.  
all_component_names = design.components.keys()  
  
gui.zoom_on_components(all_component_names)  
  
# An alternate way to view within GUI. If want to try it, remove the "#"  
       ↪from the beginning of line.  
#gui.autoscale() #resize GUI
```

3.2.4 Closing the Qiskit Metal GUI

```
[14]: gui.main_window.close()
```

```
[14]: True
```

3.2.5 My first Quantum Design (QDesign)

A Quantum Design (QDesign) can be selected from the design library `qiskit_metal.designs`. All designs are children of the `QDesign` base class, which defines the basic functionality of a `QDesign`.

We will start with the simple planar `QDesign`.

```
design = designs.DesignPlanar()
```

Interactivly view, edit, and simulate QDesign: Metal GUI To launch the qiskit metal GUI, use the method `MetalGUI`.

```
gui = MetalGUI(design)
```

```
[3]: design = designs.DesignPlanar()  
gui = MetalGUI(design)
```

3.2.6 My First Quantum Component (QComponent)

3.2.6.1 A transmon qubit

We can create a ready-made and optimized transmon qubit form the QLibrary of components. Qubit qcomponents are stored in the library `qiskit_metal qlibrary.qubits`. The

file that contains the transmon pocket is called `transmon_pocket`, and the `QComponent` class inside it is `TransmonPocket`.

- Let's create a new qubit by creating an object of this class.

```
[4]: # Select a QComponent to create (The QComponent is a python class named
      ↪ `TransmonPocket`)
from qiskit_metal qlibrary.qubits.transmon_pocket import TransmonPocket

# Create a new qcomponent object with name 'Q1'
q1 = TransmonPocket(design, 'Q1')
gui.rebuild() # rebuild the design and plot
```

```
[5]: gui.edit_component('Q1')
gui.autoscale()
```

Let's see what the Q1 object looks like

```
[6]: q1
```

```
[6]: name:    Q1
class:    TransmonPocket
options:
'pos_x'          : '0.0um',
'pos_y'          : '0.0um',
'orientation'   : '0.0',
'chip'           : 'main',
'layer'          : '1',
'connection_pads': {
},
'pad_gap'        : '30um',
'inductor_width': '20um',
'pad_width'      : '455um',
'pad_height'     : '90um',
'pocket_width'   : '650um',
'pocket_height'  : '650um',
'hfss_wire_bonds': False,
'q3d_wire_bonds' : False,
'hfss_inductance': '10nH',
'hfss_capacitance': 0,
'hfss_resistance': 0,
'hfss_mesh_kw_jj': 7e-06,
'q3d_inductance': '10nH',
'q3d_capacitance': 0,
'q3d_resistance': 0,
```

```
'q3d_mesh_kw_jj'      : 7e-06,
'gds_cell_name'        : 'my_other_junction',
module: qiskit_metal qlibrary.qubits.transmon_pocket
id:    1
```

3.2.6.1.1 What are the default options?

The QComponent comes with some default options. The options are used in the `make` function of the `qcomponent` to create the QGeometry you see in the plot above. * Options are parsed by Qiskit Metal. * You can change them from the gui or the script api.

[7]: `%metal_print` How do I edit options? API or GUI

```
<IPython.core.display.HTML object>
```

You can use the gui to create, edit, plot, modify, quantum components. Equivalently, you can also do everything form the python API. The GUI is just calling the API for you.

[8]: `# Change options`

```
q1.options.pos_x = '0.5 mm'
q1.options.pos_y = '0.25 mm'
q1.options.pad_height = '225 um'
q1.options.pad_width = '250 um'
q1.options.pad_gap = '50 um'

# Update the geoemtry, since we changed the options
gui.rebuild()
gui.autoscale()
```

3.2.6.2 Where are the QComponents stored?

They are stored in `design.components`. It can be accessed as a dictionary (`design.components['Q1']`) or object (`design.components.Q1`).

[9]: `q1 = design.components['Q1']`

[10]: `%metal_print` Where are the default options?

```
<IPython.core.display.HTML object>
```

A QComponent is created with default options. To find out what these are use `QComponentClass.get_template_options(design)`

[11]: `TransmonPocket.get_template_options(design)`

```
[11]: {'pos_x': '0.0um',
 'pos_y': '0.0um',
 'orientation': '0.0',
 'chip': 'main',
 'layer': '1',
 'connection_pads': {},
 '_default_connection_pads': {'pad_gap': '15um',
 'pad_width': '125um',
 'pad_height': '30um',
 'pad_cpw_shift': '5um',
 'pad_cpw_extent': '25um',
 'cpw_width': 'cpw_width',
 'cpw_gap': 'cpw_gap',
 'cpw_extend': '100um',
 'pocket_extent': '5um',
 'pocket_rise': '65um',
 'loc_W': '+1',
 'loc_H': '+1'},
 'pad_gap': '30um',
 'inductor_width': '20um',
 'pad_width': '455um',
 'pad_height': '90um',
 'pocket_width': '650um',
 'pocket_height': '650um',
 'hfss_wire_bonds': False,
 'q3d_wire_bonds': False,
 'hfss_inductance': '10nH',
 'hfss_capacitance': 0,
 'hfss_resistance': 0,
 'hfss_mesh_kw_jj': 7e-06,
 'q3d_inductance': '10nH',
 'q3d_capacitance': 0,
 'q3d_resistance': 0,
 'q3d_mesh_kw_jj': 7e-06,
 'gds_cell_name': 'my_other_junction'}
```

```
[12]: %metal_print How do I change the default options?
```

```
<IPython.core.display.HTML object>
```

Now lets change the default options we will use to create the transmon

```
[13]: # THIS ISN'T CHANGING THE DEFAULT OPTIONS - NEEDS UPDATE
q1.options.pos_x = '0.5 mm'
q1.options.pos_y = '250 um'
```

```
# Rebuid for changes to propagate
gui.rebuild()
```

[14]: %metal_print How do I work with units?

 (parse options and values)

<IPython.core.display.HTML object>

3.2.6.2.1 Parsing strings into floats

Use the `design.parse_value` or `QComponent.parse_value` (such as `q1.parse_value`). The two functions serve the same purpose.

```
[15]: print('Design default units for length: ', design.get_units())
print('\nExample 250 micron parsed to design units:', design.
      parse_value('0.250 um'), design.get_units())

dictionary = {'key_in_cm': '1.2 cm', 'key_in_microns': '50 um'}
print('\nExample parse dict:', design.parse_value(dictionary))

a_list = ['1m', '1mm', '1um', '1 nm']
print('\nExample parse list:', design.parse_value(a_list))
```

Design default units for length: mm

Example 250 micron parsed to design units: 0.00025 mm

Example parse dict: {'key_in_cm': 12.0, 'key_in_microns': 0.05}

Example parse list: [1000.0, 1, 0.001, 1.0000000000000002e-06]

3.2.6.2.2 Some basic arithmetic and parsing

[16]: `design.parse_value('2 * 2um')`

[16]: 0.004

[17]: `design.parse_value('2um + 5um')`

[17]: 0.007

[18]: `design.qgeometry.tables['junction']`

component	name	geometry	layer
0	1 rect_jj	LINESTRING (0.50000 0.22500, 0.50000 0.27500)	1

```

subtract helper chip width hfss_inductance hfss_capacitance \
0    False    False main 0.02          10nH      0

hfss_resistance hfss_mesh_kw_jj q3d_inductance q3d_capacitance \
0            0        0.000007       10nH      0

q3d_resistance q3d_mesh_kw_jj      gds_cell_name
0            0        0.000007 my_other_junction

```

3.2.6.3 Advanced: parse into arrays, list, etc.

Can use python syntax inside options. Parse uses pythonic `ast_eval`.

```
[19]: #### List
print('* '*10+' LIST '+'* '*10, '\n')
str_in = "[1,2,3,'10um']"
out = design.parse_value(str_in)
print(f'Parsed output:\n {str_in} -> {out} \n Out type: {type(out)}\n')

str_in = "'2*2um', '2um + 5um'"
out = design.parse_value(str_in)
print(f'Parsed output:\n {str_in} -> {out} \n Out type: {type(out)}\n')

#### Dict
print('* '*10+' DICT '+'* '*10, '\n')

str_in = "{'key1': '100um', 'key2': '1m'}"
out = design.parse_value(str_in)
print(f'Parsed output:\n {str_in} -> {out} \n Out type: {type(out)}\n')
```

* * * * * * * * * LIST * * * * * * * * *

Parsed output:
`[1,2,3,'10um'] -> [1, 2, 3, 0.01]`
Out type: <class 'list'>

Parsed output:
`['2*2um', '2um + 5um'] -> [0.004, 0.007]`
Out type: <class 'list'>

* * * * * * * * * DICT * * * * * * * * *

Parsed output:
`{'key1': '100um', 'key2': '1m'} -> {'key1': 0.1, 'key2': 1000.0}`

```
Out type: <class 'addict.addict.Dict'>
```

3.2.7 How do I overwrite QComponents?

To enable component overwrite of components with the same name, use the following cell

```
[20]: design.overwrite_enabled = True
```

```
[21]: %metal_heading Quantum pins: QPins!
```

```
<IPython.core.display.HTML object>
```

3.2.7.1 QPins: The dynamic way to connect qcomponents

The component designer can define pins. Pins can be used to link components together. For example, two transmon can each have a pin. The two pins can be connected by CPWs, as we will show below.

First, let us add pins to the transmon. We will add 4 pins called a, b, c, and d. Each pin will be at a different location (corner of the transmon), defined by the options `loc_W` and `loc_H`.

```
[22]: from qiskit_metal qlibrary qubits transmon_pocket import TransmonPocket

design.delete_all_components()

options = dict(
pad_width = '425 um',
pocket_height = '650um',
connection_pads=dict( # pin connectors
    a = dict(loc_W=+1, loc_H=+1),
    b = dict(loc_W=-1, loc_H=+1, pad_height='30um'),
    c = dict(loc_W=+1, loc_H=-1, pad_width='200um'),
    d = dict(loc_W=-1, loc_H=-1, pad_height='50um')
)
)

q1 = TransmonPocket(design, 'Q1', options = dict(pos_x='+0.5mm', pos_y='+0.5mm', **options))
```

```
[23]: # Take a screenshot with the component highlighted and the pins shown
gui.rebuild()
gui.autoscale()
gui.edit_component('Q1')
gui.zoom_on_components(['Q1'])
```

```
gui.highlight_components(['Q1'])
#gui.screenshot()
```

To access a pin

```
[24]: q1.pins.a
q1.pins['a']
```

```
[24]: {'points': [array([0.925, 0.7]), array([0.925, 0.69])],
'middle': array([0.925, 0.695]),
'normal': array([1., 0.]),
'tangent': array([0., 1.]),
'width': 0.01,
'gap': 0.006,
'chip': 'main',
'parent_name': 2,
'net_id': 0,
'length': 0}
```

3.2.7.2 How do I edit the component source code and see changes immidiately?

If you have selected a QComponent, you can call the button that says edit source in the gui. Once selected, you could also call the same function from the code.

```
[25]: gui.edit_component('Q1')
```

This will pop open a new source editor window, you can change the source on the fly. * Make sure you press the Rebuild component button in the source editor when you are ready to save and make your changes.

```
[26]: %metal_heading My first quantum chip
```

```
<IPython.core.display.HTML object>
```

3.2.8 Creating a whole chip of qubit with connectors

Let's now create a a whole chip. In the following, you will pass options to create 4 transmons qubits in a ring. First let us clear all qcomponents in the design.

```
[27]: design.delete_all_components()
gui.rebuild() # refresh
```

```
[28]: from qiskit_metal qlibrary.qubits.transmon_pocket import TransmonPocket

# Allow running the same cell here multiple times to overwrite changes
```

```

design.overwrite_enabled = True

## Custom options for all the transmons
options = dict(
# Some options we want to modify from the defaults
# (see below for defaults)
pad_width = '425 um',
pocket_height = '650um',
# Adding 4 connectors (see below for defaults)
connection_pads=dict(
    a = dict(loc_W=+1,loc_H=+1),
    b = dict(loc_W=-1,loc_H=+1, pad_height='30um'),
    c = dict(loc_W=+1,loc_H=-1, pad_width='200um'),
    d = dict(loc_W=-1,loc_H=-1, pad_height='50um')
)
)

## Create 4 transmons

q1 = TransmonPocket(design, 'Q1', options = dict(
pos_x='+2.55mm', pos_y='+0.0mm', **options))
q2 = TransmonPocket(design, 'Q2', options = dict(
pos_x='+0.0mm', pos_y=''-0.9mm', orientation = '90', **options))
q3 = TransmonPocket(design, 'Q3', options = dict(
pos_x=''-2.55mm', pos_y='+0.0mm', **options))
q4 = TransmonPocket(design, 'Q4', options = dict(
pos_x='+0.0mm', pos_y=''+0.9mm', orientation = '90', **options))

## Rebuild the design
gui.rebuild()
gui.autoscale()

```

[29]:

```
gui.toggle_docks(True)
#gui.screenshot()
```

Let's import the basic cpw QComponent from the QLibrary. It is a class called RouteMeander. We can see its default options using RouteMeander.get_template_options(design)

[30]:

```
from qiskit_metal qlibrary.tlines.meandered import RouteMeander
RouteMeander.get_template_options(design)
```

[30]:

```
{'chip': 'main',
'layer': '1',
```

```
'pin_inputs': {'start_pin': {'component': '', 'pin': ''},  

'end_pin': {'component': '', 'pin': ''}},  

'fillet': '0',  

'lead': {'start_straight': '0mm',  

'end_straight': '0mm',  

'start_jogged_extension': '',  

'end_jogged_extension': ''},  

'total_length': '7mm',  

'trace_width': 'cpw_width',  

'meander': {'spacing': '200um', 'asymmetry': '0um'},  

'snap': 'true',  

'prevent_short_edges': 'true',  

'hfss_wire_bonds': False,  

'q3d_wire_bonds': False}
```

We can now modify the options and connect all four qubits. Since this is repetitive, you can define a function to wrap up the repetatvie steps. Here we will call this `connect`. This function creates a `RouteMeander` QComponent class.

```
[31]: options = Dict(  

meander=Dict(  

    lead_start='0.1mm',  

    lead_end='0.1mm',  

    asymmetry='0 um')  

)  
  

def connect(component_name: str, component1: str, pin1: str, component2: str,  

           pin2: str,  

           length: str,  

           asymmetry='0 um', flip=False):  

    """Connect two pins with a CPW."""  

myoptions = Dict(  

    pin_inputs=Dict(  

        start_pin=Dict(  

            component=component1,  

            pin=pin1),  

        end_pin=Dict(  

            component=component2,  

            pin=pin2)),  

    lead=Dict(  

        start_straight='0.13mm'  

    ),  

    total_length=length,
```

```

fillet = '90um')
myoptions.update(options)
myoptions.meander.asymmetry = asymmetry
myoptions.meander.lead_direction_inverted = 'true' if flip else 'false'
return RouteMeander(design, component_name, myoptions)

asym = 150
cpw1 = connect('cpw1', 'Q1', 'd', 'Q2', 'c', '6.0 mm', f'{+{asym}um}')
cpw2 = connect('cpw2', 'Q3', 'c', 'Q2', 'a', '6.1 mm', f'{-{asym}um}', ↴
    flip=True)
cpw3 = connect('cpw3', 'Q3', 'a', 'Q4', 'b', '6.0 mm', f'{+{asym}um}')
cpw4 = connect('cpw4', 'Q1', 'b', 'Q4', 'd', '6.1 mm', f'{-{asym}um}', ↴
    flip=True)

gui.rebuild()
gui.autoscale()

```

[32]:

```

gui.toggle_docks(True)
gui. ↴
highlight_components(['Q1', 'Q2', 'Q3', 'Q4', 'cpw1', 'cpw2', 'cpw3', 'cpw4'])
#gui.screenshot()

```

[33]:

```
design.components.keys()
```

[33]:

```
['Q1', 'Q2', 'Q3', 'Q4', 'cpw1', 'cpw2', 'cpw3', 'cpw4']
```

We can access the created CPW from the design too.

[34]:

```
design.components.cpw2
```

```

[34]: name: cpw2
      class: RouteMeander
      options:
        'chip' : 'main',
        'layer' : '1',
        'pin_inputs' :
          'start_pin' :
            'component' : 'Q3',
            'pin' : 'c',
          },
        'end_pin' :
          'component' : 'Q2',
          'pin' : 'a',
        }
      }
    }
  }
}

```

```

        },
    },
'fillet'          : '90um',
'lead'           : {
    'start_straight'   : '0.13mm',
    'end_straight'     : '0mm',
    'start_jogged_extension': '',
    'end_jogged_extension': '',
},
'total_length'    : '6.1 mm',
'trace_width'     : 'cpw_width',
'meander'         : {
    'spacing'          : '200um',
    'asymmetry'        : '-150um',
    'lead_start'       : '0.1mm',
    'lead_end'         : '0.1mm',
    'lead_direction_inverted': 'true',
},
'snap'            : 'true',
'prevent_short_edges': 'true',
'hfss_wire_bonds' : False,
'q3d_wire_bonds'  : False,
'trace_gap'        : 'cpw_gap',
'_actual_length'   : '6.100000000000002 mm',
module: qiskit_metal qlibrary.tlines.meandered
id:      8

```

We can see all the pins

[35]: %metal_heading Variables in options

<IPython.core.display.HTML object>

3.2.9 Variables

The design can have variables, which can be used in the component options.

[36]: design.variables.cpw_width = '10um'
design.variables.cpw_gap = '6um'
gui.rebuild()

For example, we can all qubit pads using the variables.

[37]: cpw1.options.lead.end_straight = '100um'
cpw2.options.lead.end_straight = '100um'

```
cpw3.options.lead.end_straight = '100um'  
cpw4.options.lead.end_straight = '100um'
```

```
[38]: # Set variables in the design  
design.variables.pad_width = '450 um'  
design.variables.cpw_width = '25 um'  
design.variables.cpw_gap = '12 um'  
  
# Assign variables to component options  
q1.options.pad_width = 'pad_width'  
q2.options.pad_width = 'pad_width'  
q3.options.pad_width = 'pad_width'  
q4.options.pad_width = 'pad_width'  
  
# Rebuild all component and refresh the gui  
gui.rebuild()  
gui.autoscale()
```

```
[39]: %metal_heading Render to GDS
```

```
<IPython.core.display.HTML object>
```

```
[40]: gds = design.renderers.gds  
gds.options.path_filename
```

```
[40]: '../resources/Fake_Junctions.GDS'
```

```
[41]: gds.options.path_filename = '../resources/Fake_Junctions.GDS'
```

```
[42]: q1.options
```

```
[42]: {'pos_x': '+2.55mm',  
      'pos_y': '+0.0mm',  
      'orientation': '0.0',  
      'chip': 'main',  
      'layer': '1',  
      'connection_pads': {'a': {'pad_gap': '15um',  
                                'pad_width': '125um',  
                                'pad_height': '30um',  
                                'pad_cpw_shift': '5um',  
                                'pad_cpw_extent': '25um',  
                                'cpw_width': 'cpw_width',  
                                'cpw_gap': 'cpw_gap',  
                                'cpw_extend': '100um',
```

```
'pocket_extent': '5um',
'pocket_rise': '65um',
'loc_W': 1,
'loc_H': 1},
'b': {'pad_gap': '15um',
'pad_width': '125um',
'pad_height': '30um',
'pad_cpw_shift': '5um',
'pad_cpw_extent': '25um',
'cpw_width': 'cpw_width',
'cpw_gap': 'cpw_gap',
'cpw_extend': '100um',
'pocket_extent': '5um',
'pocket_rise': '65um',
'loc_W': -1,
'loc_H': 1},
'c': {'pad_gap': '15um',
'pad_width': '200um',
'pad_height': '30um',
'pad_cpw_shift': '5um',
'pad_cpw_extent': '25um',
'cpw_width': 'cpw_width',
'cpw_gap': 'cpw_gap',
'cpw_extend': '100um',
'pocket_extent': '5um',
'pocket_rise': '65um',
'loc_W': 1,
'loc_H': -1},
'd': {'pad_gap': '15um',
'pad_width': '125um',
'pad_height': '50um',
'pad_cpw_shift': '5um',
'pad_cpw_extent': '25um',
'cpw_width': 'cpw_width',
'cpw_gap': 'cpw_gap',
'cpw_extend': '100um',
'pocket_extent': '5um',
'pocket_rise': '65um',
'loc_W': -1,
'loc_H': -1},
'pad_gap': '30um',
'inductor_width': '20um',
'pad_width': 'pad_width',
'pad_height': '90um',
```

```
'pocket_width': '650um',
'pocket_height': '650um',
'hfss_wire_bonds': False,
'q3d_wire_bonds': False,
'hfss_inductance': '10nH',
'hfss_capacitance': 0,
'hfss_resistance': 0,
'hfss_mesh_kw_jj': 7e-06,
'q3d_inductance': '10nH',
'q3d_capacitance': 0,
'q3d_resistance': 0,
'q3d_mesh_kw_jj': 7e-06,
'gds_cell_name': 'my_other_junction'}
```

3.2.10 gds.options.path_filename

Need to give the correct path and filename that has gds formatted cells. An example is located in the repository under resources directory. The cells are “junctions” that would be placed inside qubits. Fake_Junctions has three cells named: “Fake_Junction_01”, “Fake_Junction_02”, “my_other_junction”.

Example: When creating transmon the default_option for gds_cell_name is “my_other_junction”.

```
[43]: gds.options.path_filename = "../resources/Fake_Junctions.GDS"
```

```
[44]: design.renderers.gds.export_to_gds("awesome_design.gds")
```

```
[44]: 1
```

Can close Metal GUI from both notebook and GUI.

```
[46]: gui.main_window.close()
```

```
[46]: True
```

3.3 Save your chip design

```
[47]: ## Design your chip
from qiskit_metal import designs, MetalGUI

design = designs.DesignPlanar()

gui = MetalGUI(design)
```

```

from qiskit_metal qlibrary.qubits.transmon_pocket_cl import_
    →TransmonPocketCL
options = {'pos_x': '0.7mm', 'pos_y': '0mm', 'connection_pads':_
    →{'readout': {'pad_gap': '15um', 'pad_width': '125um', 'pad_height':_
        →'30um', 'pad_cpw_shift': '5um', 'pad_cpw_extent': '25um', 'cpw_width':_
            →'cpw_width', 'cpw_gap': 'cpw_gap', 'cpw_extend': '100um',_
                →'pocket_extent': '5um', 'pocket_rise': '65um', 'loc_W': 1, 'loc_H': 1},_
                    →'bus': {'pad_gap': '15um', 'pad_width': '125um', 'pad_height': '30um',_
                        →'pad_cpw_shift': '5um', 'pad_cpw_extent': '25um', 'cpw_width':_
                            →'cpw_width', 'cpw_gap': 'cpw_gap', 'cpw_extend': '100um',_
                                →'pocket_extent': '5um', 'pocket_rise': '65um', 'loc_W': -1, 'loc_H':_
                                    →-1}}, 'chip': 'main', 'pad_gap': '30um', 'inductor_width': '20um',_
            →'pad_width': '425 um', 'pad_height': '90um', 'pocket_width': '650um',_
                →'pocket_height': '650um', 'orientation': '0', 'make_CL': True, 'cl_gap':_
                    →'6um', 'cl_width': '10um', 'cl_length': '20um', 'cl_ground_gap':_
                        →'6um', 'cl_pocket_edge': '180', 'cl_off_center': '50um',_
                            →'hfss_wire_bonds': False, 'q3d_wire_bonds': False, 'hfss_inductance':_
                                →'14nH', 'hfss_capacitance': 0, 'hfss_resistance': 0, 'hfss_mesh_kw_jj':_
                                    →7e-06, 'q3d_inductance': '10nH', 'q3d_capacitance': 0, 'q3d_resistance':_
                                        → 0, 'q3d_mesh_kw_jj': 7e-06, 'gds_cell_name': 'FakeJunction_01'}
Q1 = TransmonPocketCL(design, name='Q1',_
    options=options,_
    options_connection_pads={'readout': {'pad_gap': '15um', 'pad_width':_
        →'125um', 'pad_height': '30um', 'pad_cpw_shift': '5um', 'pad_cpw_extent':_
            → '25um', 'cpw_width': 'cpw_width', 'cpw_gap': 'cpw_gap', 'cpw_extend':_
                →'100um', 'pocket_extent': '5um', 'pocket_rise': '65um', 'loc_W': 1,_
                    →'loc_H': 1}, 'bus': {'pad_gap': '15um', 'pad_width': '125um',_
                        →'pad_height': '30um', 'pad_cpw_shift': '5um', 'pad_cpw_extent': '25um',_
                            →'cpw_width': 'cpw_width', 'cpw_gap': 'cpw_gap', 'cpw_extend': '100um',_
                                →'pocket_extent': '5um', 'pocket_rise': '65um', 'loc_W': -1, 'loc_H':_
                                    →-1}},_
    make=True)
Q1.meta = {}

from qiskit_metal qlibrary.qubits.transmon_pocket_cl import_
    →TransmonPocketCL

```

```

options = {'pos_x': '-0.7mm', 'pos_y': '0mm', 'connection_pads':{
    'readout': {'pad_gap': '15um', 'pad_width': '125um', 'pad_height': '30um',
                'pad_cpw_shift': '5um', 'pad_cpw_extent': '25um', 'cpw_width': 'cpw_width',
                'cpw_gap': 'cpw_gap', 'cpw_extend': '100um'},
    'pocket_extent': '5um', 'pocket_rise': '65um', 'loc_W': 1, 'loc_H': 1},
    'bus': {'pad_gap': '15um', 'pad_width': '125um', 'pad_height': '30um',
            'pad_cpw_shift': '5um', 'pad_cpw_extent': '25um', 'cpw_width': 'cpw_width',
            'cpw_gap': 'cpw_gap', 'cpw_extend': '100um'},
    'pocket_extent': '5um', 'pocket_rise': '65um', 'loc_W': -1, 'loc_H': -1}},
    'chip': 'main', 'pad_gap': '30um', 'inductor_width': '20um',
    'pad_width': '425 um', 'pad_height': '90um', 'pocket_width': '650um',
    'pocket_height': '650um', 'orientation': '180', 'make_CL': True,
    'cl_gap': '6um', 'cl_width': '10um', 'cl_length': '20um',
    'cl_ground_gap': '6um', 'cl_pocket_edge': '180', 'cl_off_center': '50um',
    'hfss_wire_bonds': False, 'q3d_wire_bonds': False,
    'hfss_inductance': '12nH', 'hfss_capacitance': 0, 'hfss_resistance': 0,
    'hfss_mesh_kw_jj': 7e-06, 'q3d_inductance': '10nH', 'q3d_capacitance': 0,
    'q3d_resistance': 0, 'q3d_mesh_kw_jj': 7e-06, 'gds_cell_name': 'FakeJunction_02'}
```

Q2 = TransmonPocketCL(design, name='Q2',
options=options,
options_connection_pads={'readout': {'pad_gap': '15um', 'pad_width': '125um', 'pad_height': '30um', 'pad_cpw_shift': '5um', 'pad_cpw_extent': '25um', 'cpw_width': 'cpw_width', 'cpw_gap': 'cpw_gap', 'cpw_extend': '100um'}, 'pocket_extent': '5um', 'pocket_rise': '65um', 'loc_W': 1, 'loc_H': 1},
'bus': {'pad_gap': '15um', 'pad_width': '125um', 'pad_height': '30um', 'pad_cpw_shift': '5um', 'pad_cpw_extent': '25um', 'cpw_width': 'cpw_width', 'cpw_gap': 'cpw_gap', 'cpw_extend': '100um'}, 'pocket_extent': '5um', 'pocket_rise': '65um', 'loc_W': -1, 'loc_H': -1},
'chip': 'main', 'pad_gap': '30um', 'inductor_width': '20um', 'pad_width': '425 um', 'pad_height': '90um', 'pocket_width': '650um', 'pocket_height': '650um', 'orientation': '180', 'make_CL': True},
'make=True)
Q2.meta = {}


```

from qiskit_metal qlibrary.tlines.pathfinder import RoutePathfinder
options = {'pin_inputs': {'start_pin': {'component': 'Q1', 'pin': 'bus'}, 'end_pin': {'component': 'Q2', 'pin': 'bus'}}, 'fillet': '99um', 'lead': {'start_straight': '0mm', 'end_straight': '250um'}, 'start_jogged_extension': '', 'end_jogged_extension': ''}, 'total_length': '7mm', 'chip': 'main', 'layer': '1', 'trace_width': 'cpw_width', 'anchors': {}, 'advanced': {'avoid_collision': 'true'}, 'step_size': '0.25mm', 'hfss_wire_bonds': False, 'q3d_wire_bonds': False, 'trace_gap': 'cpw_gap', '_actual_length': '0.8550176727053895 mm'}
```

```

Bus_Q1_Q2 = RoutePathfinder(design,
name='Bus_Q1_Q2',
options=options,
type='CPW')
Bus_Q1_Q2.meta = {}

from qiskit_metal qlibrary lumped cap_3_interdigital import
→Cap3Interdigital
options = {'layer': '1', 'trace_width': '10um', 'finger_length': '40um',□
→'pocket_buffer_width_x': '10um', 'pocket_buffer_width_y': '30um',□
→'pos_x': '2.5mm', 'pos_y': '0.25mm', 'orientation': '90'}
Cap_Q1 = Cap3Interdigital(design,
name='Cap_Q1',
options=options,
component_template=None,
make=True)
Cap_Q1.meta = {}

from qiskit_metal qlibrary lumped cap_3_interdigital import
→Cap3Interdigital
options = {'layer': '1', 'trace_width': '10um', 'finger_length': '40um',□
→'pocket_buffer_width_x': '10um', 'pocket_buffer_width_y': '30um',□
→'pos_x': '-2.5mm', 'pos_y': '-0.25mm', 'orientation': '-90'}
Cap_Q2 = Cap3Interdigital(design,
name='Cap_Q2',
options=options,
component_template=None,
make=True)
Cap_Q2.meta = {}

from qiskit_metal qlibrary tlines meandered import RouteMeander
options = {'pin_inputs': {'start_pin': {'component': 'Q1', 'pin':□
→'readout'}, 'end_pin': {'component': 'Cap_Q1', 'pin': 'a'}}, 'fillet':□
→'99um', 'lead': {'start_straight': '0.325mm', 'end_straight': '125um',□
→'start_jogged_extension': '', 'end_jogged_extension': ''}, □
→'total_length': '5mm', 'chip': 'main', 'layer': '1', 'trace_width':□
→'cpw_width', 'meander': {'spacing': '200um', 'asymmetry': '-50um'}, □
→'snap': 'true', 'prevent_short_edges': 'true', 'hfss_wire_bonds':□
→False, 'q3d_wire_bonds': False, 'trace_gap': 'cpw_gap', □
→'_actual_length': '5.000000000000001 mm'}
Readout_Q1 = RouteMeander(design,

```

```

name='Readout_Q1',
options=options,
type='CPW')
Readout_Q1.meta = {}

from qiskit_metal qlibrary.tlines.meandered import RouteMeander
options = {'pin_inputs': {'start_pin': {'component': 'Q2', 'pin': 'readout'}, 'end_pin': {'component': 'Cap_Q2', 'pin': 'a'}}, 'fillet': '99um', 'lead': {'start_straight': '0.325mm', 'end_straight': '125um', 'start_jogged_extension': '', 'end_jogged_extension': ''}, 'total_length': '6mm', 'chip': 'main', 'layer': '1', 'trace_width': 'cpw_width', 'meander': {'spacing': '200um', 'asymmetry': '-50um'}, 'snap': 'true', 'prevent_short_edges': 'true', 'hfss_wire_bonds': False, 'q3d_wire_bonds': False, 'trace_gap': 'cpw_gap', '_actual_length': '5.999999999999999 mm'}
Readout_Q2 = RouteMeander(design,
name='Readout_Q2',
options=options,
type='CPW')
Readout_Q2.meta = {}

from qiskit_metal qlibrary terminations launchpad_wb import LaunchpadWirebond
options = {'layer': '1', 'trace_width': 'cpw_width', 'trace_gap': 'cpw_gap', 'lead_length': '25um', 'pos_x': '3.5mm', 'pos_y': '0um', 'orientation': '180'}
Launch_Q1_Read = LaunchpadWirebond(design,
name='Launch_Q1_Read',
options=options,
component_template=None,
make=True)
Launch_Q1_Read.meta = {}

from qiskit_metal qlibrary terminations launchpad_wb import LaunchpadWirebond
options = {'layer': '1', 'trace_width': 'cpw_width', 'trace_gap': 'cpw_gap', 'lead_length': '25um', 'pos_x': '-3.5mm', 'pos_y': '0um', 'orientation': '0'}
Launch_Q2_Read = LaunchpadWirebond(design,
name='Launch_Q2_Read',
options=options,

```

```

component_template=None,
make=True)
Launch_Q2_Read.meta = {}

from qiskit_metal qlibrary terminations launchpad_wb import
→LaunchpadWirebond
options = {'layer': '1', 'trace_width': 'cpw_width', 'trace_gap': 'cpw_gap', 'lead_length': '25um', 'pos_x': '1.35mm', 'pos_y': '-2.5mm', 'orientation': '90'}
Launch_Q1_CL = LaunchpadWirebond(design,
name='Launch_Q1_CL',
options=options,
component_template=None,
make=True)
Launch_Q1_CL.meta = {}

from qiskit_metal qlibrary terminations launchpad_wb import
→LaunchpadWirebond
options = {'layer': '1', 'trace_width': 'cpw_width', 'trace_gap': 'cpw_gap', 'lead_length': '25um', 'pos_x': '-1.35mm', 'pos_y': '2.5mm', 'orientation': '-90'}
Launch_Q2_CL = LaunchpadWirebond(design,
name='Launch_Q2_CL',
options=options,
component_template=None,
make=True)
Launch_Q2_CL.meta = {}

from qiskit_metal qlibrary tlines pathfinder import RoutePathfinder
options = {'pin_inputs': {'start_pin': {'component': 'Launch_Q1_Read', 'pin': 'tie'}, 'end_pin': {'component': 'Cap_Q1', 'pin': 'b'}, 'fillet': '99um', 'lead': {'start_straight': '0mm', 'end_straight': '150um', 'start_jogged_extension': '', 'end_jogged_extension': ''}, 'total_length': '7mm', 'chip': 'main', 'layer': '1', 'trace_width': 'cpw_width', 'anchors': {}, 'advanced': {'avoid_collision': 'true'}, 'step_size': '0.25mm', 'hfss_wire_bonds': False, 'q3d_wire_bonds': False, 'trace_gap': 'cpw_gap', '_actual_length': '1.0750176727053897mm'}
TL_Q1 = RoutePathfinder(design,
name='TL_Q1',
options=options,

```

```

type='CPW')
TL_Q1.meta = {}

from qiskit_metal qlibrary.tlines.pathfinder import RoutePathfinder
options = {'pin_inputs': {'start_pin': {'component': 'Launch_Q2_Read', 'pin': 'tie'}, 'end_pin': {'component': 'Cap_Q2', 'pin': 'b'}}, 'fillet': '99um', 'lead': {'start_straight': '0mm', 'end_straight': '150um', 'start_jogged_extension': '', 'end_jogged_extension': ''}, 'total_length': '7mm', 'chip': 'main', 'layer': '1', 'trace_width': 'cpw_width', 'anchors': {}, 'advanced': {'avoid_collision': 'true'}, 'step_size': '0.25mm', 'hfss_wire_bonds': False, 'q3d_wire_bonds': False, 'trace_gap': 'cpw_gap', '_actual_length': '1.0750176727053897 mm'}
TL_Q2 = RoutePathfinder(design,
name='TL_Q2',
options=options,
type='CPW')
TL_Q2.meta = {}

from qiskit_metal qlibrary.tlines.pathfinder import RoutePathfinder
options = {'pin_inputs': {'start_pin': {'component': 'Launch_Q1_CL', 'pin': 'tie'}, 'end_pin': {'component': 'Q1', 'pin': 'Charge_Line'}}, 'fillet': '99um', 'lead': {'start_straight': '0mm', 'end_straight': '150um', 'start_jogged_extension': '', 'end_jogged_extension': ''}, 'total_length': '7mm', 'chip': 'main', 'layer': '1', 'trace_width': 'cpw_width', 'anchors': {}, 'advanced': {'avoid_collision': 'true'}, 'step_size': '0.25mm', 'hfss_wire_bonds': False, 'q3d_wire_bonds': False, 'trace_gap': 'cpw_gap', '_actual_length': '2.610508836352695 mm'}
TL_Q1_CL = RoutePathfinder(design,
name='TL_Q1_CL',
options=options,
type='CPW')
TL_Q1_CL.meta = {}

from qiskit_metal qlibrary.tlines.pathfinder import RoutePathfinder

```

```

options = {'pin_inputs': {'start_pin': {'component': 'Launch_Q2_CL', u
→ 'pin': 'tie'}, 'end_pin': {'component': 'Q2', 'pin': 'Charge_Line'}}, u
→ 'fillet': '99um', 'lead': {'start_straight': '0mm', 'end_straight': u
→ '150um', 'start_jogged_extension': '', 'end_jogged_extension': ''}, u
→ 'total_length': '7mm', 'chip': 'main', 'layer': '1', 'trace_width': u
→ 'cpw_width', 'anchors': {}, 'advanced': {'avoid_collision': 'true'}, u
→ 'step_size': '0.25mm', 'hfss_wire_bonds': False, 'q3d_wire_bonds': u
→ False, 'trace_gap': 'cpw_gap', '_actual_length': '2.610508836352695 mm'}
TL_Q2_CL = RoutePathfinder(design,
name='TL_Q2_CL',
options=options,
type='CPW')
TL_Q2_CL.meta = {}

gui.rebuild()

```

[48]: *## get script:*
design.to_python_script()

```

[48]: """
from qiskit_metal qlibrary.tlines.meandered import RouteMeander\n
from qiskit_metal qlibrary.tlines.pathfinder import RoutePathfinder\n
from qiskit_metal qlibrary.lumped.cap_3_interdigital import\n
    Cap3Interdigital\n
from qiskit_metal qlibrary.qubits.transmon_pocket_cl import\n
    TransmonPocketCL\n
from qiskit_metal qlibrary terminations.launchpad_wb import\n
    LaunchpadWirebond\n
from qiskit_metal import designs, MetalGUI\n
design = designs.\n    DesignPlanar()\n
design = MetalGUI(design)\n
# WARNING\n
# options_connection_pads failed to have a value\n
Q1 = TransmonPocketCL(\n    design,\n    name='Q1',\n    options={'cl_pocket_edge': u
→ '180',\n
'connection_pads': {'bus': {'cpw_extend': '100um',\n
'cpw_gap': 'cpw_gap',\n
'cpw_width': 'cpw_width',\n
'loc_H': -1,\n
'pad_cpw_extent': '25um',\n
'pad_height': '30um',\n
'pad_width': '125um',\n
'loc_W': -1,\n
'pad_cpw_shift': 'pad_cpw_shift',\n
'pad_gap': '15um',\n
'pad_width': 'pad_width'}

```



```

'orientation': '180', \n 'pad_width': '425 um', \n 'pos_x': '-0.7mm', \n
→ 'pos_y':
'0mm'}\n)\n\n\n\n\n\nBus_Q1_Q2 = RoutePathfinder(\ndesign,
\nname='Bus_Q1_Q2', \noptions={'_actual_length': '0.8550176727053895 '\n
'mm', \n 'fillet': '99um', \n 'lead': {'end_jogged_extension': '', \n
'end_straight': '250um', \n 'start_jogged_extension': '', \n
'start_straight': '0mm'}, \n 'pin_inputs': {'end_pin': {'component': \n
→ 'Q2', \n
'pin': 'bus'}, \n 'start_pin': {'component': 'Q1', \n
'pin': 'bus'}}}, \n 'trace_gap': \n
→ 'cpw_gap'}, \n\n\n\nCap_Q1 =
Cap3Interdigital(\ndesign, \nname='Cap_Q1', \noptions={'finger_length': \n
→ '40um', \n
'orientation': '90', \n 'pos_x': '2.5mm', \n 'pos_y':
'0.25mm'}, \n\n\ncomponent_template=None, \n)\n\n\n\nCap_Q2 =
Cap3Interdigital(\ndesign, \nname='Cap_Q2', \noptions={'finger_length': \n
→ '40um', \n
'orientation': '-90', \n 'pos_x': '-2.5mm', \n 'pos_y':
'-0.25mm'}, \n\n\ncomponent_template=None, \n)\n\n\n\nReadout_Q1 =
RouteMeander(\ndesign, \nname='Readout_Q1', \noptions={'_actual_length':
'5.00000000000001 '\n
'mm', \n 'fillet': '99um', \n
→ 'lead':
{'end_jogged_extension': '', \n 'end_straight': '125um', \n
'start_jogged_extension': '', \n 'start_straight': '0.325mm'}, \n
'meander': {'asymmetry': '-50um', \n 'spacing': '200um'}, \n
'pin_inputs': {'end_pin': {'component': 'Cap_Q1', \n
'pin': 'a'}, \n 'start_pin': {'component': 'Q1', \n
'pin': 'readout'}}}, \n 'total_length': '5mm', \n 'trace_gap':
'cpw_gap'}, \n\n\n\nReadout_Q2 = RouteMeander(\ndesign,
\nname='Readout_Q2', \noptions={'_actual_length': '5.999999999999999 '\n
'mm', \n 'fillet': '99um', \n 'lead': {'end_jogged_extension': '', \n
'end_straight': '125um', \n 'start_jogged_extension': ''}, \n
'start_straight': '0.325mm'}, \n 'meander': {'asymmetry': '-50um', \n
'spacing': '200um'}, \n 'pin_inputs': {'end_pin': {'component': 'Cap_Q2', \n
'pin': 'a'}, \n 'start_pin': {'component': 'Q2', \n
'pin': 'readout'}}}, \n 'total_length': '6mm', \n 'trace_gap':
'cpw_gap'}, \n\n\n\nLaunch_Q1_Read =
LaunchpadWirebond(\ndesign, \n
→ \nname='Launch_Q1_Read', \noptions={'orientation':
'180', \n 'pos_x': '3.5mm', \n 'pos_y':
'0um'}, \n\n\ncomponent_template=None, \n)\n\n\n\nLaunch_Q2_Read =
LaunchpadWirebond(\ndesign, \n
→ \nname='Launch_Q2_Read', \noptions={'orientation':

```

```

'0',\n 'pos_x': '-3.5mm',\n 'pos_y':
'0um'},\n\ncomponent_template=None,\n)\n\n\n\nLaunch_Q1_CL =
LaunchpadWirebond(\ndesign,\n
    →\nname='Launch_Q1_CL',\noptions={'orientation':
'90',\n 'pos_x': '1.35mm',\n 'pos_y':
'-2.5mm'},\n\ncomponent_template=None,\n)\n\n\n\nLaunch_Q2_CL =
LaunchpadWirebond(\ndesign,\n
    →\nname='Launch_Q2_CL',\noptions={'orientation':
'-90',\n 'pos_x': '-1.35mm',\n 'pos_y':
'2.5mm'},\n\ncomponent_template=None,\n)\n\n\n\nTL_Q1 =
RoutePathfinder(\ndesign, \nname='TL_Q1',\noptions={'_actual_length':
'1.0750176727053897 '\n
                                         'mm',\n 'fillet': '99um',\n
    →'lead':
{'end_jogged_extension': '',\n
                                         'end_straight': '150um',\n
'start_jogged_extension': '',\n
                                         'start_straight': '0mm'},\n
'pin_inputs': {'end_pin': {'component': 'Cap_Q1',\n
'pin': 'b'},\n
                                         'start_pin': {'component':\n
    →'Launch_Q1_Read',\n
'pin': 'tie'}},\n
                                         'trace_gap':\n
    →'cpw_gap'},\n\nntype='CPW',\n)\n\n\n\nTL_Q2 =
RoutePathfinder(\ndesign, \nname='TL_Q2',\noptions={'_actual_length':
'1.0750176727053897 '\n
                                         'mm',\n 'fillet': '99um',\n
    →'lead':
{'end_jogged_extension': '',\n
                                         'end_straight': '150um',\n
'start_jogged_extension': '',\n
                                         'start_straight': '0mm'},\n
'pin_inputs': {'end_pin': {'component': 'Cap_Q2',\n
'pin': 'b'},\n
                                         'start_pin': {'component':\n
    →'Launch_Q2_Read',\n
'pin': 'tie'}},\n
                                         'trace_gap':\n
    →'cpw_gap'},\n\nntype='CPW',\n)\n\n\n\nTL_Q1_CL =
RoutePathfinder(\ndesign, \nname='TL_Q1_CL',\noptions={'_actual_length':
'2.610508836352695 '\n
                                         'mm',\n 'fillet': '99um',\n
    →'lead':
{'end_jogged_extension': '',\n
                                         'end_straight': '150um',\n
'start_jogged_extension': '',\n
                                         'start_straight': '0mm'},\n
'pin_inputs': {'end_pin': {'component': 'Q1',\n
'pin': 'Charge_Line'},\n
                                         'start_pin': {'component':\n
'Launch_Q1_CL',\n
                                         'pin': 'tie'}},\n
    →'trace_gap':
'cpw_gap'},\n\nntype='CPW',\n)\n\n\n\nTL_Q2_CL =
RoutePathfinder(\ndesign,
    →\nname='TL_Q2_CL',\noptions={'_actual_length': '2.610508836352695 '\n
'mm',\n 'fillet': '99um',\n 'lead': {'end_jogged_extension': ''},\n

```

```
'end_straight': '150um',\n                 'start_jogged_extension': '',\n'start_straight': '0mm'},\n                 'pin_inputs': {'end_pin': {'component':\n                   'Q2',\n\n'pin': 'Charge_Line'},\n                 'start_pin': {'component':\n                   'Launch_Q2_CL',\n                   'pin': 'tie'}}},\n                 'trace_gap':\n'cpw_gap'},\n                 ntype='CPW',\n)\n\n\nngui.rebuild()\nngui.autoscale()\n\n"\n\n
```

[49]: *## Copy the script printed above here. Comment out all other code and run.*

[50]: `gui.main_window.close()`

[50]: `True`

[]:

3.4 Routing Between QComponents

3.4.0.1 Introduction

Routes are strips of metal (or cuts in the bulk metal), that electrically connect two input-specified pins (pin = point on the perimeter of a QComponent, with orientation, indicating an allowed point for electrical contact).

The core class `QRoute` is designed to support different route types, currently only single or double (CPW) wiring. We will use the CPW transmission line in the remainder of this notebook, and in the majority of the other tutorial notebooks.

`QRoute` inherits the generic `QComponent`. `QRoute` also has two attributes of type `QRouteLead`, which enable close control of the start and end points in a route. We will describe in this notebook how to utilize the `QRouteLead.QRoutePoint` is a convenient exchange format for directed points.

`QRoute` is an abstract class, without a `make()` method, thus cannot be instantiated as a design component. `QRoute` is inherited by classes that can be instantiated (have a `make()` method - blue highlights in the image below). By subclassing further, you can implement comprehensive routing algorithms, such as the `RouteMixed`.

In this notebook we will only look at the simple `RouteStraight` and `RouteLead`.

3.4.0.2 Prerequisite

Initialize metal

```
[1]: %reload_ext autoreload  
%autoreload 2
```

```
[2]: from qiskit_metal import designs  
from qiskit_metal import MetalGUI, Dict  
  
design = designs.DesignPlanar()  
gui = MetalGUI(design)  
  
design.variables['cpw_width'] = '10 um'  
design.variables['cpw_gap'] = '6 um'
```

```
[3]: # enable rebuild of the same component  
design.overwrite_enabled = True
```

We will use the following classes in the notebook. Let's import all of them at once.

```
[4]: # for pins:  
from qiskit_metal qlibrary terminations open_to_ground import OpenToGround  
# for routing:
```

```
from qiskit_metal qlibrary.tlines.straight_path import RouteStraight
from qiskit_metal qlibrary.tlines.framed_path import RouteFramed
# for jogs:
from collections import OrderedDict
```

To remove the complexity of large QComponents, we will only use the `OpenToGround` QComponent to establish pins. We want to initialize a dictionary of type `Dict` as follows (naming matching Example 1). We will later update this dictionary as needed using the simpler dot notation.

```
[5]: pin_opt = Dict(pin_inputs=Dict(start_pin=Dict(
                                         component='open1i',
                                         pin='open'),
                                         end_pin=Dict(
                                         component='open1o',
                                         pin='open')))
```

3.4.1 Example 1: Straight routing between two pins

First, we make sure our design contains two pins to connect to.

```
[6]: design.delete_all_components() #needed only for rebuilds. will get a
→warning

otg11 = OpenToGround(design, 'open1i', options=dict(pos_x='0mm', □
→pos_y='1mm', orientation='180'))
otg12 = OpenToGround(design, 'open1o', options=dict(pos_x='0.5mm', □
→pos_y='1mm', orientation='0'))

gui.rebuild()
gui.autoscale()
```

Then we create the route in between them. Notice we are using the previously defined `pin_opt` to indicate which pins to connect to.

```
[7]: route1 = RouteStraight(design, 'cpw_1', pin_opt)

gui.rebuild()
gui.autoscale()
gui.highlight_components(['open1i', 'open1o'])
#gui.screenshot()
```

3.4.2 Example 2: Any direction

Let's add a few pins that we can use to connect in pairs. We here align them following a variety of orientations.

```
[8]: otg21 = OpenToGround(design, 'open2i', options=dict(pos_x='0mm', □
    ↪pos_y='2mm', orientation='90'))
otg22 = OpenToGround(design, 'open2o', options=dict(pos_x='0mm', □
    ↪pos_y='1.5mm', orientation='-90'))
otg31 = OpenToGround(design, 'open3i', options=dict(pos_x='0.35mm', □
    ↪pos_y='1.55mm', orientation='45'))
otg32 = OpenToGround(design, 'open3o', options=dict(pos_x='0mm', □
    ↪pos_y='1.2mm', orientation=-135))
otg41 = OpenToGround(design, 'open4i', options=dict(pos_x='0mm', □
    ↪pos_y='0.8mm', orientation='135'))
otg42 = OpenToGround(design, 'open4o', options=dict(pos_x='0.35mm', □
    ↪pos_y='0.45mm', orientation=-45))

gui.rebuild()
gui.autoscale()
```

Then we connect pair of pins using a straight route. Notice how we update the `pin_opt` component names to the new pair, for each new `RouteStraight`. Indeed, we can only use a pin for one single connection, and we will get an error if we try to connect a second route to the same pin.

```
[9]: pin_opt.pin_inputs.start_pin.component = 'open2i'
pin_opt.pin_inputs.end_pin.component = 'open2o'
route2 = RouteStraight(design, 'cpw_2', pin_opt)
pin_opt.pin_inputs.start_pin.component = 'open3i'
pin_opt.pin_inputs.end_pin.component = 'open3o'
route3 = RouteStraight(design, 'cpw_3', pin_opt)
pin_opt.pin_inputs.start_pin.component = 'open4i'
pin_opt.pin_inputs.end_pin.component = 'open4o'
route4 = RouteStraight(design, 'cpw_4', pin_opt)

gui.rebuild()
gui.autoscale()
```

3.4.3 Example 3: Angles and leads

If the two pins to connect are not facing each other, the router will create the necessary jogs. Here is a simple example that shows 45° jogs.

The `RouteStraight` will use a single straight line in any orientation to connect the two pins.

Feel free to try using other algorithms, by replacing the class to any other `QRoute` subclass (image at the start of this notebook).

```
[10]: otg51 = OpenToGround(design, 'open5i', options=dict(pos_x='0.7mm', ↪pos_y='1.9mm', orientation='180'))
otg52 = OpenToGround(design, 'open5o', options=dict(pos_x='0.8mm', ↪pos_y='2.0mm', orientation='90'))

pin_opt.pin_inputs.start_pin.component = 'open5i'
pin_opt.pin_inputs.end_pin.component = 'open5o'
route5 = RouteStraight(design, 'cpw_5', pin_opt)

gui.rebuild()
gui.zoom_on_components(['cpw_5'])
#gui.screenshot()
```

Let's make another connection. This time, we add a lead extension to both pins. An extension is a route segment that begins from the pin coordinates and extends straight in the pin direction by the given length.

```
[11]: otg61 = OpenToGround(design, 'open6i', options=dict(pos_x='0.7mm', ↪pos_y='1.7mm', orientation='180'))
otg62 = OpenToGround(design, 'open6o', options=dict(pos_x='0.8mm', ↪pos_y='1.8mm', orientation='90'))

pin_opt.pin_inputs.start_pin.component = 'open6i'
pin_opt.pin_inputs.end_pin.component = 'open6o'

pin_opt.lead.start_straight = '0.04mm'
pin_opt.lead.end_straight = '0.06mm'
route6 = RouteStraight(design, 'cpw_6', pin_opt)

gui.rebuild()
gui.zoom_on_components(['cpw_6'])
#gui.screenshot()
```

3.4.4 Example 4: 90° angles

In most applications, you might want to limit the jogs to 90° angles. In the following example we use the same `pin_opts` of the previous example, but we replace `RouteStraight` with `RouteFramed`. You will notice that our route now has only one 90° jog.

```
[12]:
```

```

otg71 = OpenToGround(design, 'open7i', options=dict(pos_x='0.7mm', □
    ↪pos_y='1.5mm', orientation='180'))
otg72 = OpenToGround(design, 'open7o', options=dict(pos_x='0.8mm', □
    ↪pos_y='1.6mm', orientation='90'))

pin_opt.pin_inputs.start_pin.component = 'open7i'
pin_opt.pin_inputs.end_pin.component = 'open7o'

route7 = RouteFramed(design, 'cpw_7', pin_opt)

gui.rebuild()
gui.zoom_on_components(['cpw_7'])
#gui.screenshot()

```

The route resulting from the cell above, had no trace of the lead segments. That is because the length of the lead segments was less than the edge of the corner. By extending the `start_straight` for example, we can see the effect of the lead again.

```
[14]: otg81 = OpenToGround(design, 'open8i', options=dict(pos_x='0.7mm', □
    ↪pos_y='1.3mm', orientation='180'))
otg82 = OpenToGround(design, 'open8o', options=dict(pos_x='0.8mm', □
    ↪pos_y='1.4mm', orientation='90'))

pin_opt.pin_inputs.start_pin.component = 'open8i'
pin_opt.pin_inputs.end_pin.component = 'open8o'

pin_opt.lead.start_straight = '0.15mm'
pin_opt.lead.end_straight = '0.05mm'
route8 = RouteFramed(design, 'cpw_8', pin_opt)

gui.rebuild()
gui.zoom_on_components(['cpw_8'])
```

Notice how the `RouteFramed` executed in the previous cell needs to create 2 jogs to complete the routing, as opposed to the previous single 90° job example.

This algorithm helps also connecting pins not facing each other, like in the next cell.

```
[15]: otg91 = OpenToGround(design, 'open9i', options=dict(pos_x='0.7mm', □
    ↪pos_y='1.05mm', orientation='180'))
otg92 = OpenToGround(design, 'open9o', options=dict(pos_x='0.8mm', □
    ↪pos_y='1.15mm', orientation='270'))

pin_opt.pin_inputs.start_pin.component = 'open9i'
pin_opt.pin_inputs.end_pin.component = 'open9o'
```

```
route9 = RouteFramed(design, 'cpw_9', pin_opt)

gui.rebuild()
gui.zoom_on_components(['cpw_9'])
#gui.screenshot()
```

The RouteFramed can create up to 3 segments. Observe the following case of opposing pins.

```
[16]: otg101 = OpenToGround(design, 'open10i', options=dict(pos_x='0.7mm',
    ↪pos_y='0.8mm', orientation='90'))
otg102 = OpenToGround(design, 'open10o', options=dict(pos_x='0.8mm',
    ↪pos_y='0.9mm', orientation='270'))

pin_opt.pin_inputs.start_pin.component = 'open10i'
pin_opt.pin_inputs.end_pin.component = 'open10o'

pin_opt.lead.start_straight = '0.03mm'
pin_opt.lead.end_straight = '0.03mm'
route10 = RouteFramed(design, 'cpw_10', pin_opt)

gui.rebuild()
gui.zoom_on_components(['cpw_10'])
#gui.screenshot()
```

3.4.5 Example 5: Tight control on leads and angles

QRouteLeads are a “stack” of points that describe in detail the “last-mile” to the pins where the route terminates. These stacks are seeded with the pin coordinates and direction and build incrementally.

In the previous examples we controlled the QRouteLead points by utilizing the `lead` options inside the `pin_opt` construct. Specifically, we utilized `start_straight` and `end_straight`, which add one point to the respective leads, at the specified distance in the direction specified by the pins (outwards from the QComponent that has the pin).

We are however not limited to a single aligned extension point. Utilizing the `start_jogged_extension` and `end_jogged_extension`, we can specify any number of points to add to the lead stacks. The points are computed by providing a `OrderedDict()` of `steps`, expressed as `[angle,distance]` pairs. * `distance` is the string describing how long the step needs to be * `angle` describe the direction in which to take the step

The `angle` can be described in several ways, to accommodate different coding styles. For example, all of the options above will lead to the same 90° left turn: > “L”, “L90”, “R-90”, 90, “90”, “A,90”, “left”, “left90”, “right-90”

```
[17]: otg111 = OpenToGround(design, 'open11i', options=dict(pos_x='0.7mm', □
    ↪pos_y='0.5mm', orientation='180'))
otg112 = OpenToGround(design, 'open11o', options=dict(pos_x='0.8mm', □
    ↪pos_y='0.6mm', orientation='270'))

pin_opt.pin_inputs.start_pin.component = 'open11i'
pin_opt.pin_inputs.end_pin.component = 'open11o'

# the first step is always straight, let's define by how much (minimum is □
    ↪half the route width):
pin_opt.lead.start_straight = '0.03mm'
pin_opt.lead.end_straight = '0.04mm'

# any subsequent step of the lead_start
jogsS = OrderedDict()
jogsS[0] = ["L", '20um']
jogsS[1] = ["R", '50um']
jogsS[2] = [90, '30um']
jogsS[3] = [-90, '60um']
jogsS[4] = ["90", '40um']
jogsS[5] = ["-90", '70um']
jogsS[6] = ["L30", '30um']
jogsS[7] = ["A,30", '30um']
jogsS[8] = ["left30", '50um']

# single jog on the lead_end, just for kicks
jogsE = OrderedDict()
jogsE[0] = ["L", '30um']

pin_opt.lead.start_jogged_extension = jogsS
pin_opt.lead.end_jogged_extension = jogsE
route10 = RouteFramed(design, 'cpw_11', pin_opt)

gui.rebuild()
gui.zoom_on_components(['cpw_11'])
#gui.screenshot()
```

[18]: gui.autoscale()

[19]: gui.main_window.close()

[19]: True

3.5 Simple Meander

Creates 3 transmon pockets, connected in different ways by the mean of transmission lines.

These transmission line QComponents create basic meanders to accommodate the user-defined line length.

3.5.0.1 Preparations

```
[1]: %reload_ext autoreload
%autoreload 2
```

Import key libraries and open the Metal GUI. Also we configure the notebook to enable overwriting of existing components

```
[2]: from qiskit_metal import designs
from qiskit_metal import MetalGUI, Dict

design = designs.DesignPlanar()
gui = MetalGUI(design)

# if you disable the next line, then you will need to delete a component ↴
→ [<component>.delete()] before recreating it
design.overwrite_enabled = True
```

```
[3]: from qiskit_metal qlibrary.core import QRouteLead
```

Create 3 Transmon Qbits with 4 pins. This uses the same definition (options) for all 3 Qbits, but it places them in 3 different (x,y) origin points.

```
[4]: from qiskit_metal qlibrary.qubits.transmon_pocket import TransmonPocket

optionsQ = dict(
    pad_width = '425 um',
    pocket_height = '650um',
    connection_pads = dict( # Qbits defined to have 4 pins
        a = dict(loc_W=+1, loc_H=+1),
        b = dict(loc_W=-1, loc_H=+1, pad_height='30um'),
        c = dict(loc_W=+1, loc_H=-1, pad_width='200um'),
        d = dict(loc_W=-1, loc_H=-1, pad_height='50um')
    )
)

q1 = TransmonPocket(design, 'Q1', options = dict(pos_x=' -1.5mm', ↴
→ pos_y=' +0.0mm', **optionsQ))
```

```

q2 = TransmonPocket(design, 'Q2', options = dict(pos_x='+0.35mm', □
    ↪pos_y='+1.0mm', orientation = '90',**optionsQ))
q3 = TransmonPocket(design, 'Q3', options = dict(pos_x='2.0mm', pos_y='+0.□
    ↪0mm', **optionsQ))

gui.rebuild()
gui.autoscale()
gui.highlight_components(['Q1', 'Q2', 'Q3']) # This is to show the pins, □
    ↪so we can choose what to connect

```

3.6 Using CPW meanders to connect the 3 Qbits

Import the RouteMeander and inspect what options are available for you to initialize

```
[5]: from qiskit_metal qlibrary.tlines.meandered import RouteMeander
RouteMeander.get_template_options(design)
```

```
[5]: {'chip': 'main',
'layer': '1',
'pin_inputs': {'start_pin': {'component': '', 'pin': ''},
'end_pin': {'component': '', 'pin': ''}},
'fillet': '0',
'lead': {'start_straight': '0mm',
'end_straight': '0mm',
'start_jogged_extension': '',
'end_jogged_extension': ''},
'total_length': '7mm',
'trace_width': 'cpw_width',
'meander': {'spacing': '200um', 'asymmetry': '0um'},
'snap': 'true',
'prevent_short_edges': 'true',
'hfss_wire_bonds': False,
'q3d_wire_bonds': False}
```

Let's define a dictionary with the options we want to keep global. We will then concatenate this dictionary to the ones we will use to describe the individual Qbits

```
[6]: ops=dict(fillet='90um')
```

Let's define the first CPW QComponent.

```
[7]: options = Dict(
    total_length= '8mm',
    hfss_wire_bonds = True,
```

```

pin_inputs=Dict(
    start_pin=Dict(
        component= 'Q1',
        pin= 'a'),
    end_pin=Dict(
        component= 'Q2',
        pin= 'b')),
lead=Dict(
    start_straight='0mm',
    end_straight='0.5mm'),
meander=Dict(
    asymmetry='-1mm'),
**ops
)

# Below I am creating a CPW without assigning its name.
# Therefore running this cell twice will create two CPW's instead of ↴
# →overwriting the previous one
# To prevent that we add the cpw.delete() statement.
# The try-except wrapping is needed to suppress errors during the first ↴
# →run of this cell
try:
    cpw.delete()
except NameError: pass

cpw = RouteMeander(design, options=options)
gui.rebuild()
gui.autoscale()

```

11:21PM 22s WARNING [check_lengths]: For path table, component=cpw_1, ↴
 ↪key=trace
 has short segments that could cause issues with fillet. Values in (1-1) are
 index(es) in shapely geometry.

11:21PM 22s WARNING [check_lengths]: For path table, component=cpw_1, ↴
 ↪key=cut
 has short segments that could cause issues with fillet. Values in (1-1) are
 index(es) in shapely geometry.

11:21PM 22s WARNING [check_lengths]: For path table, component=cpw_1, ↴
 ↪key=trace
 has short segments that could cause issues with fillet. Values in (1-1) are
 index(es) in shapely geometry.

11:21PM 22s WARNING [check_lengths]: For path table, component=cpw_1, ↴
 ↪key=cut
 has short segments that could cause issues with fillet. Values in (1-1) are

index(es) in shapely geometry.

You might have received an expected python “warning - check_lengths” message. This indicates that one of the CPW edges is too short to accommodate the fillet corner rounding previously defined at 90um. Also, the CPW “start lead” does not offer enough clearance from the Qbit, causing a short between two pins.

Please take a minute to observe this behavior in the GUI

You can inspect the points forming the CPW route to find the culprit edge (the warning message should have indicated the index of the offending edge)

```
[8]: cpw.get_points()
```

```
[8]: array([[-1.075      ,  0.195      ] ,  
          [-1.07       ,  0.195      ] ,  
          [-1.07       , -0.44916667] ,  
          [-0.87       , -0.44916667] ,  
          [-0.87       , -1.29421683] ,  
          [-0.67       , -1.29421683] ,  
          [-0.67       , -0.31578317] ,  
          [-0.47       , -0.31578317] ,  
          [-0.47       , -1.29421683] ,  
          [-0.27       , -1.29421683] ,  
          [-0.27       , -0.31578317] ,  
          [-0.07       , -0.31578317] ,  
          [-0.07       , -1.29421683] ,  
          [ 0.155      , -1.29421683] ,  
          [ 0.155      ,  0.575      ] ] )
```

Turns out that both issues can be resolved by just adding enough clearance at the start of the CPW.

```
[9]: cpw.options['lead']['start_straight']='100um'  
gui.rebuild()  
gui.autoscale()
```

Notice how the routing algorithm tries to prevent creation of edges too short to apply the fillet corner rounding. If your design requires it, you can disable this algorithm behavior for individual CPWs. For example try the cell below, and expect warning similar to the ones encountered earlier to show up

```
[10]: cpw.options['prevent_short_edges']='false'  
gui.rebuild()  
gui.autoscale()
```

```
11:21PM 23s WARNING [check_lengths]: For path table, component=cpw_1,
  ↵key=trace
has short segments that could cause issues with fillet. Values in (12-13) ↵
  ↵are
index(es) in shapely geometry.
11:21PM 23s WARNING [check_lengths]: For path table, component=cpw_1,
  ↵key=cut
has short segments that could cause issues with fillet. Values in (12-13) ↵
  ↵are
index(es) in shapely geometry.
```

You can eliminate the warning by either re-enabling the algorithm, or by changing the fillet for this specific CPW. Let's try the second approach for demonstration purposes

```
[11]: cpw.options['fillet']='65um'
gui.rebuild()
gui.autoscale()
```

Now let's create other 3 CPWs for practice

```
[12]: options = Dict(
    total_length= '6mm',
    pin_inputs=Dict(
        start_pin=Dict(
            component= 'Q2',
            pin= 'd'),
        end_pin=Dict(
            component= 'Q3',
            pin= 'b')),
    lead=Dict(
        start_straight='0.1mm',
        end_straight='0.2mm'),
    meander=Dict(
        asymmetry='-0.9mm'),
    **ops
)

try:
    cpw2.delete()
except NameError: pass

cpw2 = RouteMeander(design,options=options)
gui.rebuild()
gui.autoscale()
```

```
[13]: options = Dict(
    total_length= '8mm',
    pin_inputs=Dict(
        start_pin=Dict(
            component= 'Q3',
            pin= 'a'),
        end_pin=Dict(
            component= 'Q2',
            pin= 'c')),
    lead=Dict(
        start_straight='0.5mm',
        end_straight='0.1mm'),
    meander=Dict(
        asymmetry='-1mm'),
    **ops
)

try:
    cpw3.delete()
except NameError: pass

cpw3 = RouteMeander(design,options=options)
gui.rebuild()
gui.autoscale()
```

```
[14]: options = Dict(
    total_length= '8mm',
    pin_inputs=Dict(
        start_pin=Dict(
            component= 'Q1',
            pin= 'b'),
        end_pin=Dict(
            component= 'Q2',
            pin= 'a')),
    lead=Dict(
        start_straight='0.5mm',
        end_straight='0.1mm'),
    meander=Dict(
        asymmetry='1mm'),
    **ops
)

try:
    cpw4.delete()
```

```
except NameError: pass

cpw4 = RouteMeander(design,options=options)
gui.rebuild()
gui.autoscale()
```

Let's try a more complex utilization of the leads. So far we have demonstrated `lead->start_straight` and `lead->end_straight`. We can "append" to the straight lead any number of custom jogs, which might be useful to get out of complex layout arrangements or to fine tune the meander in case of collisions with other components.

To define a jogged lead, you need an ordered sequence of turn-length pairs, which we define here as an ordered dictionary. We then apply the sequence of jogs to both the start and end leads (or you could apply it to only one them, or you can define the two leads separately with two ordered dictionaries)

```
[15]: from collections import OrderedDict
jogs = OrderedDict()
jogs[0] = ["L", '800um']
jogs[1] = ["L", '500um']
jogs[2] = ["R", '200um']
jogs[3] = ["R", '500um']

options = Dict(
    total_length= '14mm',
    pin_inputs=Dict(
        start_pin=Dict(
            component= 'Q1',
            pin= 'd'),
        end_pin=Dict(
            component= 'Q3',
            pin= 'd')),
    lead=Dict(
        start_straight='0.1mm',
        end_straight='0.1mm',
        start_jogged_extension=jogs,
        end_jogged_extension=jogs),
    meander=Dict(
        asymmetry='-1.2mm'),
    **ops
)

try:
    cpw5.delete()
except NameError: pass
```

```
cpw5 = RouteMeander(design,options=options)
gui.rebuild()
gui.autoscale()
```

Here a few additional examples routing flexibility on a brand-new set of Qbits

```
[16]: q4 = TransmonPocket(design, 'Q4', options = dict(pos_x='-7.5mm',  

    →pos_y='-0.5mm', **optionsQ))
q5 = TransmonPocket(design, 'Q5', options = dict(pos_x='-5.65mm',  

    →pos_y='+0.5mm', orientation = '90',**optionsQ))
q6 = TransmonPocket(design, 'Q6', options = dict(pos_x='-4.0mm',  

    →pos_y='-0.6mm', **optionsQ))
gui.rebuild()

options = Dict(
    total_length= '3.4mm',
    pin_inputs=Dict(
        start_pin=Dict(
            component= 'Q4',
            pin= 'a'),
        end_pin=Dict(
            component= 'Q5',
            pin= 'b')),
    lead=Dict(
        start_straight='0.5mm',
        end_straight='0.1mm'),
    meander=Dict(
        asymmetry='1mm'),
    **ops
)
try:
    cpw6.delete()
except NameError: pass
cpw6 = RouteMeander(design, options=options)
gui.rebuild()
gui.autoscale()
```

```
[17]: options = Dict(
    total_length= '12mm',
    pin_inputs=Dict(
        start_pin=Dict(
            component= 'Q4',
            pin= 'd'),
```

```
end_pin=Dict(
    component= 'Q6',
    pin= 'c')),
lead=Dict(
    start_straight='0.1mm',
    end_straight='0.1mm'),
meander=Dict(
    asymmetry='-1.2mm'),
**ops
)
try:
    cpw7.delete()
except NameError: pass
cpw7 = RouteMeander(design, options=options)
gui.rebuild()
gui.autoscale()
```

[18] :

```
options = Dict(
    total_length= '13mm',
    pin_inputs=Dict(
        start_pin=Dict(
            component= 'Q6',
            pin= 'a'),
        end_pin=Dict(
            component= 'Q4',
            pin= 'b')),
    lead=Dict(
        start_straight='0.1mm',
        end_straight='0.1mm'),
    meander=Dict(
        asymmetry='-1.7mm'),
    **ops
)
try:
    cpw8.delete()
except NameError: pass
cpw8 = RouteMeander(design, options=options)
gui.rebuild()
gui.autoscale()
```

[19] : `gui.main_window.close()`

[19] : True

3.7 Hybrid Auto and AStar

Creates 3 transmon pockets in an L shape, each of which can be rotated in increments of 90 deg. Anchors are user-specified points through which the CPW must pass. For a specified step size and suitable choice of anchors, a snapped path can always be found. How close this path is to the shortest path depends on the step size - a smaller step size generally yields more optimal paths but requires a longer runtime.

3.7.1 Preparations

```
[20]: %load_ext autoreload  
%autoreload 2
```

The autoreload extension is already loaded. To reload it, use:
 %reload_ext autoreload

Import key libraries and open the Metal GUI. Also we configure the notebook to enable overwriting of existing components

```
[21]: from qiskit_metal import designs, draw  
from qiskit_metal import MetalGUI, Dict  
  
design = designs.DesignPlanar()  
gui = MetalGUI(design)  
  
# If you disable the next line, then you will need to delete a component  
# →[<component>.delete()] before recreating it.  
design.overwrite_enabled = True
```

Create 3 transmon qubits with 4 pins. This uses the same definition (options) for all 3 qubits, but it places them in 3 different (x,y) origin points.

```
[22]: from qiskit_metal qlibrary.qubits.transmon_pocket import TransmonPocket  
  
options = dict(  
    pad_width = '425 um',  
    pocket_height = '650um',  
    connection_pads=dict( # pin connectors  
        a = dict(loc_W=+1,loc_H=+1),  
        b = dict(loc_W=-1,loc_H=+1, pad_height='30um'),  
        c = dict(loc_W=+1,loc_H=-1, pad_width='200um'),  
        d = dict(loc_W=-1,loc_H=-1, pad_height='50um')  
    )  
)
```

```

q0 = TransmonPocket(design, 'Q0', options = dict(pos_x='-1.0mm', pos_y='1.0mm', **options))
q1 = TransmonPocket(design, 'Q1', options = dict(pos_x='1.0mm', pos_y='+0.0mm', **options))
q2 = TransmonPocket(design, 'Q2', options = dict(pos_x='1.0mm', pos_y='0.0mm', **options))

gui.rebuild()
gui.autoscale()

```

Import the RoutePathfinder class and inspect what options are available for you to initialize.

```
[23]: from qiskit_metal qlibrary.tlines.pathfinder import RoutePathfinder
RoutePathfinder.get_template_options(design)
```

```
[23]: {'chip': 'main',
'layer': '1',
'pin_inputs': {'start_pin': {'component': '', 'pin': ''},
'end_pin': {'component': '', 'pin': ''}},
'fillet': '0',
'lead': {'start_straight': '0mm',
'end_straight': '0mm',
'start_jogged_extension': '',
'end_jogged_extension': ''},
'total_length': '7mm',
'trace_width': 'cpw_width',
'anchors': {},
'advanced': {'avoid_collision': 'true'},
'step_size': '0.25mm',
'hfss_wire_bonds': False,
'q3d_wire_bonds': False}
```

Here we've set the fillet radius to be 90 microns.

```
[24]: ops=dict(fillet='90um')
```

Using the 3-qubit arrangement from before, our goal is to connect pins on two of them. Moreover, we want some degree of control over how that path is constructed. This is where anchors come into play. Anchors are predefined points in space that the path must cross. They are stored in an ordered dictionary data structure that maps the anchor number to a numpy array of length 2. The anchor number is an integer, starting at 0, that determines that anchor point's position relative to the start pin; larger anchor numbers are farther along the path. The numpy array stores the x and y coordinates of the anchor.

For this and the subsequent examples, we have chosen a step size of 0.25 mm, which means that the algorithm searches for valid paths to the next anchor point in increments of that

amount. Valid paths do not collide with other components or self-intersect. The start and end pins are labelled by their component and component-specific pin, in this case Q0_b and Q1_b.

```
[25]: import numpy as np
from collections import OrderedDict

anchors = OrderedDict()
anchors[0] = np.array([0.048, -0.555])
anchors[1] = np.array([0.048, 0.195])

options = {'pin_inputs':
            {'start_pin': {'component': 'Q0', 'pin': 'b'},
             'end_pin': {'component': 'Q1', 'pin': 'b'}},
            'lead': {'start_straight': '91um', 'end_straight': '90um'},
            'step_size': '0.25mm',
            'anchors': anchors,
            **ops
           }
          }

qa = RoutePathfinder(design, 'line', options)

gui.rebuild()
gui.autoscale()
```

The current algorithm can only allow 1 such CPW to be built at a time, thus we delete old CPWs before building new ones.

```
[26]: design.delete_component('line')

gui.rebuild()
```

Now let's try wrapping the CPW around the outer edge of the bottom left qubit.

```
[27]: anchors = OrderedDict()
anchors[0] = np.array([-0.452, -0.555])
anchors[1] = np.array([-0.452, -1.5])
anchors[2] = np.array([0.048, -1.5])

options = {'pin_inputs':
            {'start_pin': {'component': 'Q0', 'pin': 'b'},
             'end_pin': {'component': 'Q1', 'pin': 'b'}},
            'lead': {'start_straight': '91um', 'end_straight': '90um'},
            'step_size': '0.25mm',
            'anchors': anchors,
```

```
    **ops
}

qa = RoutePathfinder(design, 'line', options)

gui.rebuild()
gui.autoscale()
```

```
[28]: design.delete_component('line')
```

```
gui.rebuild()
```

We can also try switching around the components/pins and not specifying any anchor points to see what happens.

```
[29]: options = {'pin_inputs':
    {'start_pin': {'component': 'Q0', 'pin': 'a'},
     'end_pin': {'component': 'Q2', 'pin': 'd'}},
    'lead': {'start_straight': '90um', 'end_straight': '90um'},
    'step_size': '0.25mm',
    **ops
}

qa = RoutePathfinder(design, 'line', options)
```

```
gui.rebuild()
gui.autoscale()
```

```
[30]: design.delete_component('line')
```

```
gui.rebuild()
```

Or we can specify even more anchor points!

```
[31]: anchors = OrderedDict()
anchors[0] = np.array([-2, 0.5])
anchors[1] = np.array([0, 0.5])
anchors[2] = np.array([0, -1])
anchors[3] = np.array([2, -1])

options = {'pin_inputs':
    {'start_pin': {'component': 'Q0', 'pin': 'd'},
     'end_pin': {'component': 'Q1', 'pin': 'c'}},
    'step_size': '0.25mm',
    'anchors': anchors,
```

```

    **ops
}

qc = RoutePathfinder(design, 'line', options)

gui.rebuild()
gui.autoscale()

```

[32]: `gui.main_window.close()`

[32]: `True`

3.8 Get them all with MixedRoute

Creates 2 transmon pockets facing each other.

Anchors are user-specified points through which the Routing must pass.

Between anchors you can specify different connection algorythms.

[1]: `%reload_ext autoreload`
`%autoreload 2`

[2]: `from qiskit_metal import designs`
`from qiskit_metal import MetalGUI, Dict`

`design = designs.DesignPlanar()`
`gui = MetalGUI(design)`

[3]: `# enable rebuild of the same component`
`design.overwrite_enabled = True`
`design.delete_all_components()`

[4]: `design.delete_all_components() #needed only for rebuilds. will get a ↴warning`
`from qiskit_metal.qlibrary.qubits.transmon_pocket import TransmonPocket`

`optionsQ = dict(`
 `pad_width = '425 um',`
 `pocket_height = '650um',`
 `connection_pads=dict(# pin connectors`
 `a = dict(loc_W=+1, loc_H=+1),`
 `b = dict(loc_W=-1, loc_H=+1, pad_height='30um'),`
 `c = dict(loc_W=+1, loc_H=-1, pad_width='200um'),`

```

        d = dict(loc_W=-1, loc_H=-1, pad_height='50um')
    )
)

q0 = TransmonPocket(design, 'Q0', options = dict(pos_x='-5.0mm', pos_y='0.
→0mm', **optionsQ))
q1 = TransmonPocket(design, 'Q1', options = dict(pos_x='5.0mm', pos_y='0.
→0mm', **optionsQ))

gui.rebuild()
gui.autoscale()

```

3.8.0.0.1 Single CPW using one meander and 3 simple segments

[5]:

```

from qiskit_metal qlibrary.tlines.mixed_path import RouteMixed
import numpy as np
from collections import OrderedDict
ops=dict(fillet='90um')

```

[6]:

```

anchors = OrderedDict()
anchors[0] = np.array([-3., 1.])
anchors[1] = np.array([0, 2])
anchors[2] = np.array([3., 1])
anchors[3] = np.array([4., .5])

between_anchors = OrderedDict() # S, M, PF
between_anchors[0] = "S"
between_anchors[1] = "M"
between_anchors[2] = "S"
between_anchors[3] = "M"
between_anchors[4] = "S"

jogsS = OrderedDict()
jogsS[0] = ["R", '200um']
jogsS[1] = ["R", '200um']
jogsS[2] = ["L", '200um']
jogsS[3] = ["L", '500um']
jogsS[4] = ["R", '200um']

jogsE = OrderedDict()
jogsE[0] = ["L", '200um']
jogsE[1] = ["L", '200um']
jogsE[2] = ["R", '200um']
jogsE[3] = ["R", '500um']

```

```
jogsE[4] = ["L", '200um']

optionsR = {'pin_inputs': {
    'start_pin': {'component': 'Q0', 'pin': 'b'},
    'end_pin': {'component': 'Q1', 'pin': 'a'}
},
    'total_length': '32mm',
    'chip': 'main',
    'layer': '1',
    'trace_width': 'cpw_width',
    'step_size': '0.25mm',
    'anchors': anchors,
    'between_anchors': between_anchors,
    'advanced': {'avoid_collision': 'true'},
    'meander': {
        'spacing': '200um',
        'asymmetry': '0um'
    },
    'snap': 'true',
    'lead': {
        'start_straight': '0.3mm',
        'end_straight': '0.3mm',
        'start_jogged_extension': jogsS,
        'end_jogged_extension': jogsE
    },
    **ops
}
}

qa = RouteMixed(design, 'line', optionsR)

gui.rebuild()
gui.autoscale()
```

3.8.0.0.2 Single CPW using the pathfinder segments

```
[7]: anchors = OrderedDict()
anchors[0] = np.array([-3., -1.])
anchors[1] = np.array([0.2, -2])
anchors[2] = np.array([3., -1.7])

between_anchors = OrderedDict() # S, M, PF
between_anchors[0] = "S"
between_anchors[1] = "S"
```

```

between_anchors[2] = "PF"
between_anchors[3] = "S"

jogsS = OrderedDict()
jogsS[0] = ["L", '200um']
jogsS[1] = ["L", '200um']
jogsS[2] = ["R", '200um']
jogsS[3] = ["R", '500um']
jogsS[4] = ["L", '200um']

jogsE = OrderedDict()
jogsE[0] = ["R", '200um']
jogsE[1] = ["R", '200um']
jogsE[2] = ["L", '200um']
jogsE[3] = ["L", '500um']
jogsE[4] = ["R", '200um']


optionsR = {'pin_inputs': {
    'start_pin': {'component': 'Q0', 'pin': 'd'},
    'end_pin': {'component': 'Q1', 'pin': 'c'}
},
    'total_length': '22mm',
    'chip': 'main',
    'layer': '1',
    'trace_width': 'cpw_width',
    'step_size': '0.25mm',
    'anchors': anchors,
    'between_anchors': between_anchors,
    'advanced': {'avoid_collision': 'true'},
    'meander': {
        'spacing': '200um',
        'asymmetry': '0um'
    },
    'snap': 'true',
    'lead': {
        'start_straight': '0.3mm',
        'end_straight': '0.3mm',
        'start_jogged_extension': jogsS,
        'end_jogged_extension': jogsE
    },
    **ops
}

```

```
qb = RouteMixed(design, 'line2', optionsR)

gui.rebuild()
gui.autoscale()
```

[8] : # create an obstacle

```
TransmonPocket(design, options = dict(pos_x='1.0mm', pos_y='-2.0mm', ↴**optionsQ))
TransmonPocket(design, options = dict(pos_x='1.0mm', pos_y='-1.0mm', ↴**optionsQ))
TransmonPocket(design, options = dict(pos_x='-2.0mm', pos_y='-1.0mm', ↴**optionsQ))

gui.rebuild()
gui.autoscale()
```

[10] : anchors = OrderedDict()
anchors[0] = np.array([0., -1.5])

between_anchors = OrderedDict() # S, M, PF
between_anchors[0] = "S"
between_anchors[1] = "PF"

jogsS = OrderedDict()
jogsS[0] = ["R", '200um']
jogsS[1] = ["R", '200um']
jogsS[2] = ["L", '200um']
jogsS[3] = ["L", '500um']

jogsE = OrderedDict()
jogsE[0] = ["L", '200um']
jogsE[1] = ["L", '200um']
jogsE[2] = ["R", '200um']
jogsE[3] = ["R", '500um']

optionsR = {'pin_inputs': {
 'start_pin': {'component': 'Q0', 'pin': 'c'},
 'end_pin': {'component': 'Q1', 'pin': 'd'}
 },
 'total_length': '12mm',
 'chip': 'main',
 'layer': '1',
 'trace_width': 'cpw_width',
 'step_size': '0.25mm',

```
'anchors': anchors,
'between_anchors': between_anchors,
'advanced': {'avoid_collision': 'true'},
'meander': {
    'spacing': '200um',
    'asymmetry': '0um'
},
'snap': 'true',
'lead': {
    'start_straight': '0.3mm',
    'end_straight': '0.3mm',
    'start_jogged_extension': jogsS,
    'end_jogged_extension': jogsE
},
**ops
}

qc = RouteMixed(design, 'line3', optionsR)

gui.rebuild()
gui.autoscale()
```

[11]: gui.main_window.close()

[11]: True

[]:

3.9 Capacitance matrix and LOM analysis

3.9.0.1 Prerequisite

You need to have a working local installation of Ansys.

3.9.1 1. Create the design in Metal

```
[1]: %reload_ext autoreload
%autoreload 2

[2]: import qiskit_metal as metal
from qiskit_metal import designs, draw
from qiskit_metal import MetalGUI, Dict, Headings

[3]: design = designs.DesignPlanar()
gui = MetalGUI(design)

from qiskit_metal qlibrary.qubits.transmon_pocket import TransmonPocket
from qiskit_metal qlibrary.tlines.meandered import RouteMeander

[4]: design.variables['cpw_width'] = '15 um'
design.variables['cpw_gap'] = '9 um'
```

3.9.1.1 In this example, the design consists of 4 qubits and 4 CPWs

```
[5]: # Allow running the same cell here multiple times to overwrite changes
design.overwrite_enabled = True

## Custom options for all the transmons
options = dict(
    # Some options we want to modify from the defaults
    # (see below for defaults)
    pad_width = '425 um',
    pocket_height = '650um',
    # Adding 4 connectors (see below for defaults)
    connection_pads=dict(
        readout = dict(loc_W=+1,loc_H=-1, pad_width='200um'),
        bus1 = dict(loc_W=-1,loc_H=+1, pad_height='30um'),
        bus2 = dict(loc_W=-1,loc_H=-1, pad_height='50um')
    )
)

## Create 4 transmons
```

```

q1 = TransmonPocket(design, 'Q1', options = dict(
    pos_x='+2.42251mm', pos_y='+0.0mm', **options))
q2 = TransmonPocket(design, 'Q2', options = dict(
    pos_x='+0.0mm', pos_y='-.95mm', orientation = '270', **options))
q3 = TransmonPocket(design, 'Q3', options = dict(
    pos_x='-.42251mm', pos_y='+0.0mm', orientation = '180', **options))
q4 = TransmonPocket(design, 'Q4', options = dict(
    pos_x='+0.0mm', pos_y='+0.95mm', orientation = '90', **options))

RouteMeander.get_template_options(design)

options = Dict(
    lead=Dict(
        start_straight='0.2mm',
        end_straight='0.2mm'),
    trace_gap='9um',
    trace_width='15um')

def connect(component_name: str, component1: str, pin1: str, component2: str,
           pin2: str,
           length: str, asymmetry='0 um', flip=False, fillet='90um'):
    """Connect two pins with a CPW."""
    myoptions = Dict(
        fillet=fillet,
        hfss_wire_bonds = True,
        pin_inputs=Dict(
            start_pin=Dict(
                component=component1,
                pin=pin1),
            end_pin=Dict(
                component=component2,
                pin=pin2)),
        total_length=length)
    myoptions.update(options)
    myoptions.meander.asymmetry = asymmetry
    myoptions.meander.lead_direction_inverted = 'true' if flip else
    ↪'false'
    return RouteMeander(design, component_name, myoptions)

asym = 140
cpw1 = connect('cpw1', 'Q1', 'bus2', 'Q2', 'bus1', '6.0 mm', f'+{asym}um')
cpw2 = connect('cpw2', 'Q3', 'bus1', 'Q2', 'bus2', '6.1 mm', ↪
    ↪f'-{asym}um', flip=True)

```

```
cpw3 = connect('cpw3', 'Q3', 'bus2', 'Q4', 'bus1', '6.0 mm', f'{asym}um')
cpw4 = connect('cpw4', 'Q1', 'bus1', 'Q4', 'bus2', '6.1 mm', ↵
    f'{asym}um', flip=True)

gui.rebuild()
gui.autoscale()
```

3.9.2 2. Capacitance Analysis and LOM derivation using the analysis package - most users

3.9.2.1 Capacitance Analysis

Select the analysis you intend to run from the `qiskit_metal.analyses` collection. Select the design to analyze and the tool to use for any external simulation

```
[6]: from qiskit_metal.analyses.quantization import LOManalysis
c1 = LOManalysis(design, "q3d")
```

(optional) You can review and update the Analysis default setup following the examples in the next two cells.

```
[7]: c1.sim.setup
```

```
[7]: {'name': 'Setup',
      'reuse_selected_design': True,
      'reuse_setup': True,
      'freq_ghz': 5.0,
      'save_fields': False,
      'enabled': True,
      'max_passes': 15,
      'min_passes': 2,
      'min_converged_passes': 2,
      'percent_error': 0.5,
      'percent_refinement': 30,
      'auto_increase_solution_order': True,
      'solution_order': 'High',
      'solver_type': 'Iterative'}
```

```
[8]: # example: update single setting
c1.sim.setup.max_passes = 6
# example: update multiple settings
c1.sim.setup_update(solution_order = 'Medium', ↵
    auto_increase_solution_order = 'False')

c1.sim.setup
```

```
[8]: {'name': 'Setup',
      'reuse_selected_design': True,
      'reuse_setup': True,
      'freq_ghz': 5.0,
      'save_fields': False,
      'enabled': True,
      'max_passes': 6,
      'min_passes': 2,
      'min_converged_passes': 2,
      'percent_error': 0.5,
      'percent_refinement': 30,
      'auto_increase_solution_order': 'False',
      'solution_order': 'Medium',
      'solver_type': 'Iterative'}
```

Analyze a single qubit with 2 endcaps using the default (or edited) analysis setup. Then show the capacitance matrix (from the last pass).

You can use the method `run()` instead of `sim.run()` in the following cell if you want to run both cap extraction and lom analysis in a single step. If so, make sure to also tweak the setup for the lom analysis. The input parameters are otherwise the same for the two methods.

```
[9]: c1.sim.run(components=['Q1'], open_terminations=[('Q1', 'readout'),
                                                     ('Q1', 'bus1'), ('Q1', 'bus2')])
c1.sim.capacitance_matrix
```

```
INFO 12:15AM [connect_project]: Connecting to Ansys Desktop API...
INFO 12:15AM [load_ansys_project]:      Opened Ansys App
INFO 12:15AM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0
INFO 12:15AM [load_ansys_project]:      Opened Ansys Project
    Folder:   C:/Users/Bartu/Documents/Ansoft/
    Project:  Project1
INFO 12:15AM [connect_design]: No active design found (or error getting
    ↪active
design).
INFO 12:15AM [connect]:           Connected to project "Project1". No design
detected
INFO 12:15AM [connect_design]:  Opened active design
    Design:  Design_q3d [Solution type: Q3D]
WARNING 12:15AM [connect_setup]:      No design setup detected.
WARNING 12:15AM [connect_setup]:      Creating Q3D default setup.
INFO 12:15AM [get_setup]:      Opened setup `Setup` (<class
'pyEPR.ansys.AnsysQ3DSetup'&)
INFO 12:15AM [get_setup]:      Opened setup `Setup` (<class
```

```
'pyEPR.ansys.AnsysQ3DSetup'>
INFO 12:15AM [analyze]: Analyzing setup Setup
INFO 12:16AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpbz6ugtva.txt, C, , Setup:LastAdaptive,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 1, False
INFO 12:16AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp4s1fvs3o.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 1, False
INFO 12:16AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpgir373_d.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 2, False
INFO 12:16AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpodyb0hou.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 3, False
INFO 12:16AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp0u6t_yk.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 4, False
INFO 12:16AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp3rkekg62.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 5, False
INFO 12:16AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpw7kafyj4.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 6, False
INFO 12:16AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpxyhd6adj.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 7, False
```

[9] :

	bus1_connector_pad_Q1	bus2_connector_pad_Q1	\	
bus1_connector_pad_Q1	49.77798	-0.42560		
bus2_connector_pad_Q1	-0.42560	54.01892		
ground_main_plane	-33.50861	-35.77523		
pad_bot_Q1	-1.57029	-13.99742		
pad_top_Q1	-13.15538	-1.82852		
readout_connector_pad_Q1	-0.20494	-1.01319		
	ground_main_plane	pad_bot_Q1	pad_top_Q1	\
bus1_connector_pad_Q1	-33.50861	-1.57029	-13.15538	
bus2_connector_pad_Q1	-35.77523	-13.99742	-1.82852	
ground_main_plane	237.69027	-31.53488	-37.88739	
pad_bot_Q1	-31.53488	98.20668	-30.07381	
pad_top_Q1	-37.88739	-30.07381	87.85084	
readout_connector_pad_Q1	-36.55733	-18.86490	-2.20122	
	readout_connector_pad_Q1			

bus1_connector_pad_Q1	-0.20494
bus2_connector_pad_Q1	-1.01319
ground_main_plane	-36.55733
pad_bot_Q1	-18.86490
pad_top_Q1	-2.20122
readout_connector_pad_Q1	59.92353

(optional - case-dependent) If the previous cell was interrupted due to license limitations and for any reason you finally manually launched the simulation from the renderer GUI (outside qiskit-metal) you might be able to recover the simulation results by uncommenting and executing the following cell

```
[10]: #c1.sim._get_results_from_renderer()
#c1.sim.capacitance_matrix
```

The last variables you pass to the `run()` or `sim.run()` methods, will be stored in the `sim.setup` dictionary under the key `run`. You can recall the information passed by either accessing the dictionary directly, or by using the print handle below.

```
[11]: # c1.setup.run      <- direct access
c1.sim.print_run_args()
```

This analysis object run with the following kwargs:

```
{'name': None, 'components': ['Q1'], 'open_terminations': [('Q1', 'readout'),
('Q1', 'bus1'), ('Q1', 'bus2')], 'box_plus_buffer': True}
```

You can re-run the analysis after varying the parameters. Not passing the parameter `components` to the `sim.run()` method, skips the rendering and tries to run the analysis on the latest design. If a design is not found, the full metal design is rendered.

```
[12]: c1.sim.setup.freq_ghz = 4.8
c1.sim.run()
c1.sim.capacitance_matrix
```

```
INFO 12:18AM [get_setup]:      Opened setup `Setup`  (<class
'pyEPR.ansys.AnsysQ3DSetup'>)
INFO 12:18AM [analyze]: Analyzing setup Setup
INFO 12:19AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpht12clgu.txt, C, , Setup:LastAdaptive,
"Original", "ohm", "nH", "fF", "mSie", 4800000000, Maxwell, 1, False
INFO 12:19AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp080u9pda.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 4800000000, Maxwell, 1, False
INFO 12:19AM [get_matrix]: Exporting matrix data to
```

```
(C:\Users\Bartu\AppData\Local\Temp\tmprluf_xho.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 4800000000, Maxwell, 2, False
INFO 12:19AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmppp1hy74r.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 4800000000, Maxwell, 3, False
INFO 12:19AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpogk5k_kd.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 4800000000, Maxwell, 4, False
INFO 12:19AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp94nh7kfz.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 4800000000, Maxwell, 5, False
INFO 12:19AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpp9j52lp4.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 4800000000, Maxwell, 6, False
INFO 12:19AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpi1095f_6.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 4800000000, Maxwell, 7, False
```

[12] :

	bus1_connector_pad_Q1	bus2_connector_pad_Q1	\
bus1_connector_pad_Q1	49.77798	-0.42560	
bus2_connector_pad_Q1	-0.42560	54.01892	
ground_main_plane	-33.50862	-35.77523	
pad_bot_Q1	-1.57029	-13.99741	
pad_top_Q1	-13.15538	-1.82852	
readout_connector_pad_Q1	-0.20494	-1.01319	
			\
bus1_connector_pad_Q1	-33.50862	-1.57029	-13.15538
bus2_connector_pad_Q1	-35.77523	-13.99741	-1.82852
ground_main_plane	237.69029	-31.53487	-37.88739
pad_bot_Q1	-31.53487	98.20668	-30.07382
pad_top_Q1	-37.88739	-30.07382	87.85083
readout_connector_pad_Q1	-36.55733	-18.86490	-2.20122
			\
			readout_connector_pad_Q1
bus1_connector_pad_Q1		-0.20494	
bus2_connector_pad_Q1		-1.01319	
ground_main_plane		-36.55733	
pad_bot_Q1		-18.86490	
pad_top_Q1		-2.20122	
readout_connector_pad_Q1		59.92354	

[13] : `type(c1.sim.capacitance_matrix)`

[13] : pandas.core.frame.DataFrame

3.9.2.2 Lumped oscillator model (LOM)

Using capacitance matrices obtained from each pass, save the many parameters of the Hamiltonian of the system. `get_lumped_oscillator()` operates on 4 setup parameters: `Lj`: float
`Cj`: float `fr`: Union[list, float] `fb`: Union[list, float]

```
[14]: c1.setup.junctions = Dict({'Lj': 12.31, 'Cj': 2})
c1.setup.freq_readout = 7.0
c1.setup.freq_bus = [6.0, 6.2]

c1.run_lom()
c1.lumped_oscillator_all
```

[3, 4] [5 0 1]

Predicted Values

Transmon Properties

f_Q 5.424936 [GHz]
EC 311.976968 [MHz]
EJ 13.273404 [GHz]
alpha -363.792432 [MHz]
dispersion 46.550381 [KHz]
Lq 12.305036 [nH]
Cq 62.088648 [fF]
T1 35.336776 [us]

Coupling Properties

tCqbus1 7.383746 [fF]
gbus1_in_MHz 114.265742 [MHz]
χ_bus1 -3.174189 [MHz]
1/T1bus1 2809.275546 [Hz]
T1bus1 56.653376 [us]

tCqbus2 -6.455283 [fF]
gbus2_in_MHz -85.831837 [MHz]
χ_bus2 -9.970480 [MHz]
1/T1bus2 1205.470756 [Hz]
T1bus2 132.027212 [us]

tCqbus3 5.372189 [fF]
gbus3_in_MHz 73.765144 [MHz]
χ_bus3 -4.515414 [MHz]

1/T1bus3 489.200061 [Hz]

T1bus3 325.337128 [us]

Bus-Bus Couplings

gbus1_2 7.097123 [MHz]

gbus1_3 9.957495 [MHz]

gbus2_3 5.377146 [MHz]

[14] : fQ EC EJ alpha dispersion \

1	5.748489	353.281616	13.273404	-417.442119	135.370322
2	5.664673	342.29546	13.273404	-403.045756	103.898361
3	5.574019	330.639542	13.273404	-387.872739	77.330698
4	5.523032	324.186391	13.273404	-379.516673	65.212227
5	5.463771	316.778222	13.273404	-369.962566	53.275088
6	5.424936	311.976968	13.273404	-363.792432	46.550381

				gbus \
1	[108.7995027684672,	-73.52728948139995,	76.416...	
2	[112.4840344536537,	-82.16155872148896,	68.802...	
3	[111.33609808483656,	-84.09228828439241,	71.77...	
4	[110.9979473053141,	-84.10820800421689,	72.871...	
5	[113.07917197513633,	-84.87713113746445,	72.79...	
6	[114.26574159060948,	-85.8318374534719,	73.765...	

			chi_in_MHz	xr MHz	gr MHz
1	[-4.794392460535969,	-26.86080064467838,	-12.4...	4.794392	108.799503
2	[-4.459428523665049,	-22.01884264095459,	-7.62...	4.459429	112.484034
3	[-3.780464271017631,	-15.865610473945122,	-6.3...	3.780464	111.336098
4	[-3.472075994127398,	-13.185863881932118,	-5.6...	3.472076	110.997947
5	[-3.2937202482767276,	-11.011716535764679,	-4...	3.293720	113.079172
6	[-3.1741887432362756,	-9.970479730018141,	-4.5...	3.174189	114.265742

[15] : c1.plot_convergence();
 c1.plot_convergence_chi()

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:503: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

 self._hfss_variables[variation] = pd.Series(

INFO 12:25AM [hfss_report_full_convergence]: Creating report for variation 0

Design "Design_q3d" info:
 # eigenmodes 0
 # variations 1

Once you are done with your analysis, please close it with `close()`. This will free up resources currently occupied by qiskit-metal to communicate with the tool.

[16]: `c1.sim.close()`

3.9.3 3. Directly access the renderer to modify other parameters

[17]: `c1.sim.start()`
`c1.sim.renderer`

```
INFO 12:26AM [connect_project]: Connecting to Ansys Desktop API...
INFO 12:26AM [load_ansys_project]:      Opened Ansys App
INFO 12:26AM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0
INFO 12:26AM [load_ansys_project]:      Opened Ansys Project
    Folder:  C:/Users/Bartu/Documents/Ansoft/
    Project: Project1
INFO 12:26AM [connect_design]:  Opened active design
    Design:  Design_q3d [Solution type: Q3D]
INFO 12:26AM [get_setup]:      Opened setup `Setup` (<class
'pyEPR.ansys.AnsysQ3DSetup'&)
INFO 12:26AM [connect]:      Connected to project "Project1" and design
"Design_q3d"
```

[17]: `<qiskit_metal.renderers.renderer_ansys.q3d_renderer.QQ3DRenderer at 0x138a6330280>`

Every renderer will have its own collection of methods. Below an example with q3d

Prepare and run a collection of predefined setups This is equivalent to going to the Project Manager panel in Ansys, right clicking on Analysis within the active Q3D design, selecting “Add Solution Setup...”, and choosing/entering default values in the resulting popup window. You might want to do this to keep track of different solution setups, giving each of them a different/specific name.

[18]: `setup = c1.sim.renderer.new_ansys_setup(name = "Setup_demo", max_passes = 6)`

You can directly pass to `new_ansys_setup` all the setup parameters. Of course you will then need to run the individual setups by name as well.

[19]: `c1.sim.renderer.analyze_setup(setup.name)`

```
INFO 12:26AM [get_setup]:      Opened setup `Setup_demo` (<class
'pyEPR.ansys.AnsysQ3DSetup'>)
```

INFO 12:26AM [analyze]: Analyzing setup Setup_demo

Get the capacitance matrix at a different pass You might want to use this if you intend to know what was the matrix at a different pass of the simulation.

```
[20]: # Using the analysis results, get its capacitance matrix as a dataframe.
c1.sim.renderer.get_capacitance_matrix(variation = '', solution_kind =
    'AdaptivePass', pass_number = 5)
```

INFO 12:26AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpzu7wau80.txt, C, ,
Setup_demo:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 5, False

```
[20]: (   bus1_connector_pad_Q1  bus2_connector_pad_Q1  \
bus1_connector_pad_Q1          48.92803           -0.42235
bus2_connector_pad_Q1          -0.42235            53.06961
ground_main_plane             -33.01607           -35.14374
pad_bot_Q1                   -1.53861           -13.67515
pad_top_Q1                   -12.84517           -1.81368
readout_connector_pad_Q1      -0.20677           -0.99928

                           ground_main_plane  pad_bot_Q1  pad_top_Q1  \
bus1_connector_pad_Q1          -33.01607       -1.53861     -12.84517
bus2_connector_pad_Q1          -35.14374       -13.67515     -1.81368
ground_main_plane              234.01567      -31.50558     -37.64437
pad_bot_Q1                    -31.50558       96.95568     -29.44070
pad_top_Q1                    -37.64437      -29.44070      86.62922
readout_connector_pad_Q1      -36.15710      -18.48102     -2.19686

                           readout_connector_pad_Q1
bus1_connector_pad_Q1          -0.20677
bus2_connector_pad_Q1          -0.99928
ground_main_plane              -36.15710
pad_bot_Q1                     -18.48102
pad_top_Q1                     -2.19686
readout_connector_pad_Q1      59.14496 ,
'fF')
```

3.9.3.1 Code to swap rows and columns in capacitance matrix

```
from qiskit_metal.analyses.quantization.lumped_capacitive import
df_reorder_matrix_basis

df_reorder_matrix_basis(fourq_q3d.get_capacitance_matrix(), 1, 2)
```



Close the renderer

[21] : `c1.sim.close()`

[] :

3.10 Eigenmode and EPR analysis

3.10.0.1 Prerequisite

You need to have a working local installation of Ansys.

3.10.1 Sections

3.10.1.1 I. Transmon only

1. Prepare the layout in qiskit-metal.
2. Run finite element eigenmode analysis.
3. Plot fields and display them.
4. Set up EPR junction dictionary.
5. Run EPR analysis on single mode.
6. Get qubit freq and anharmonicity.
7. Calculate EPR of substrate.
8. (Extra: Calculate surface EPR.)

3.10.1.2 II. Resonator only

1. Update the layout in qiskit-metal.
2. Run finite element eigenmode analysis.
3. Plot fields and display them.
4. Calculate EPR of substrate.

3.10.1.3 III. Transmon & resonator

1. Update the layout in qiskit-metal.
2. Run finite element eigenmode analysis.
3. Plot fields and display them.
4. Set up EPR junction dictionary.
5. Run EPR analysis on the two modes.
6. Get qubit frequency and anharmonicity.

3.10.1.4 IV. Analyze a coupled 2 transmon system.

1. Finite Element Eigenmode Analysis
2. Identify the mode you want. The mode can inclusively be from 1 to setup.n_modes.
3. Set variables in the Ansys design. As before, we seek 2 modes.
4. Set up the simulation and specify the variables for the sweep.
5. Plot the E-field on the chip's surface.
6. Specify the junctions in the model; in this case there are 2 junctions.
7. Find the electric and magnetic energy stored in the substrate and the system as a whole.
8. Perform EPR analysis for all modes and variations.

```
[1]: %reload_ext autoreload
%autoreload 2

import qiskit_metal as metal
from qiskit_metal import designs, draw
from qiskit_metal import MetalGUI, Dict, Headings
import pyEPR as epr
```

3.11 1. Analyze the transmon qubit by itself

We will use the analysis package - applicable to most users. Advanced users might want to expand the package, or directly interact with the renderer. The renderer is one of the properties of the analysis class.

3.11.0.1 Create the Qbit design

Setup a design of a given dimension. Dimensions will be respected in the design rendering. Note that the design size extends from the origin into the first quadrant.

```
[2]: design = designs.DesignPlanar({}, True)
design.chips.main.size['size_x'] = '2mm'
design.chips.main.size['size_y'] = '2mm'

gui = MetalGUI(design)
```

Create a single transmon with one readout resonator and move it to the center of the chip previously defined.

```
[3]: from qiskit_metal qlibrary.qubits.transmon_pocket import TransmonPocket

design.delete_all_components()

q1 = TransmonPocket(design, 'Q1', options = dict(
    pad_width = '425 um',
    pocket_height = '650um',
    connection_pads=dict(
        readout = dict(loc_W=+1, loc_H=+1, pad_width='200um')
    )))
    )

gui.rebuild()
gui.autoscale()
```

3.11.0.2 Finite Element Eigenmode Analysis

3.11.0.2.1 Setup

Select the analysis you intend to run from the `qiskit_metal.analyses` collection. Select the design to analyze and the tool to use for any external simulation.

```
[4]: from qiskit_metal.analyses.quantization import EPRanalysis
```

```
[5]: eig_qb = EPRanalysis(design, "hfss")
```

Review and update the convergence parameters and junction properties by executing following two cells. We exemplify three different methods to update the setup parameters.

```
[6]: eig_qb.sim.setup
```

```
[6]: {'name': 'Setup',
      'reuse_selected_design': True,
      'reuse_setup': True,
      'min_freq_ghz': 1,
      'n_modes': 1,
      'max_delta_f': 0.5,
      'max_passes': 10,
      'min_passes': 1,
      'min_converged': 1,
      'pct_refinement': 30,
      'basis_order': 1,
      'vars': {'Lj': '10 nH', 'Cj': '0 fF'}}
```

```
[7]: # example: update single setting
eig_qb.sim.setup.max_passes = 6
eig_qb.sim.setup.vars.Lj = '11 nH'
# example: update multiple settings
eig_qb.sim.setup_update(max_delta_f = 0.4, min_freq_ghz = 1.1)

eig_qb.sim.setup
```

```
[7]: {'name': 'Setup',
      'reuse_selected_design': True,
      'reuse_setup': True,
      'min_freq_ghz': 1.1,
      'n_modes': 1,
      'max_delta_f': 0.4,
      'max_passes': 6,
      'min_passes': 1,
      'min_converged': 1,
```

```
'pct_refinement': 30,
'basis_order': 1,
'vars': {'Lj': '11 nH', 'Cj': '0 fF'}}
```

3.11.0.2.2 Execute simulation and verify convergence and EM field

Analyze a single qubit with shorted terminations. Then observe the frequency convergence plot. If not converging, you might want to increase the min_passes value to force the renderer to increase accuracy.

You can use the method `run()` instead of `sim.run()` in the following cell if you want to run both eigenmode and epr analysis in a single step. If so, make sure to also tweak the setup for the epr analysis. The input parameters are otherwise the same for the two methods.

```
[8]: eig_qb.sim.run(name="Qbit", components=['Q1'], open_terminations=[],  
                   ↪box_plus_buffer = False)  
eig_qb.sim.plot_convergences()
```

```
INFO 10:28PM [connect_project]: Connecting to Ansys Desktop API...
INFO 10:28PM [load_ansys_project]:      Opened Ansys App
INFO 10:28PM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0
INFO 10:28PM [load_ansys_project]:      Opened Ansys Project
    Folder:   C:/Users/Bartu/Documents/Ansoft/
    Project:  Project1
INFO 10:28PM [connect_design]: No active design found (or error getting ↪
    ↪active
design).
INFO 10:28PM [connect]:           Connected to project "Project1". No design
detected
INFO 10:28PM [connect_design]:  Opened active design
    Design:   Qbit_hfss [Solution type: Eigenmode]
WARNING 10:28PM [connect_setup]:      No design setup detected.
WARNING 10:28PM [connect_setup]:      Creating eigenmode default setup.
INFO 10:28PM [get_setup]:      Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
INFO 10:28PM [get_setup]:      Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
INFO 10:28PM [analyze]: Analyzing setup Setup
10:28PM 45s INFO [get_f_convergence]: Saved convergences to
C:\Users\Bartu\Desktop\qiskit-metal\tutorials\4 Analysis\A. Core - EM and
quantization\hfss_eig_f_convergence.csv
```

The last variables you pass to the `run()` or `sim.run()` methods, will be stored in the `sim.setup` dictionary under the key `run`. You can recall the information passed by either accessing the dictionary directly, or by using the print handle below.

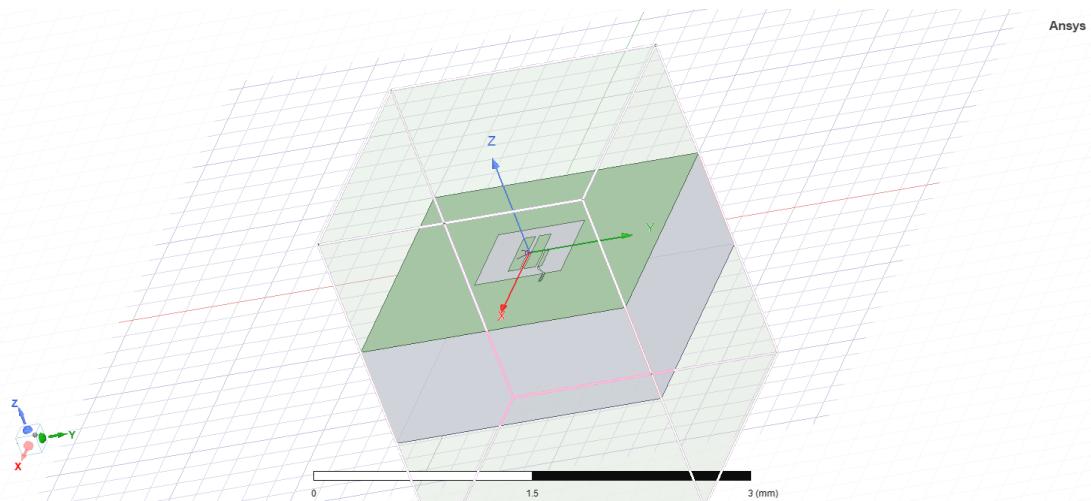
```
[9]: # eig_qb.setup.run    <- direct access
eig_qb.sim.print_run_args()
```

This analysis object run with the following kwargs:

```
{'name': 'Qbit', 'components': ['Q1'], 'open_terminations': [], 'port_list': None, 'jj_to_port': None, 'ignored_jjs': None, 'box_plus_buffer': False}
```

(optional) Captures the renderer GUI

```
[10]: eig_qb.sim.save_screenshot()
```



```
[10]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/A.u
                  ↪Core -  

                  EM and quantization/ansys.png')
```

(optional) Work directly with the convergence numbers

```
[11]: eig_qb.sim.convergence_f
```

```
[11]: re(Mode(1)) [g]
Pass []
1          3.579716
2          5.168308
3          5.923488
4          6.106727
5          6.198069
6          6.251130
```

(optional) You can re-run the analysis after varying the parameters. Not passing the param-

eter components to the `sim.run()` method, skips the rendering and tries to run the analysis on the latest design. If a design is not found, the full metal design is rendered.

```
[12]: eig_qb.sim.setup.min_freq_ghz = 4
eig_qb.sim.run()
eig_qb.sim.convergence_f
```

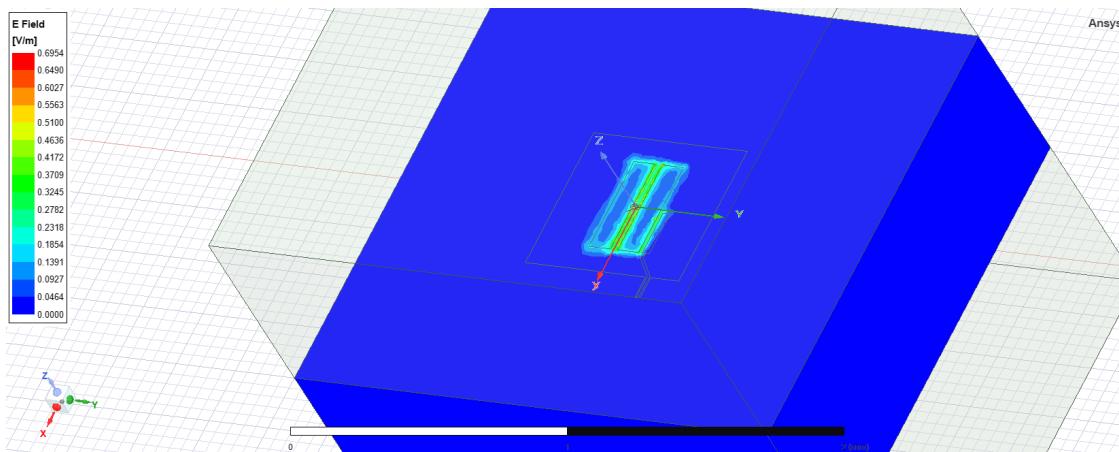
```
INFO 10:55PM [get_setup]:      Opened setup `Setup`  (<class
'pyEPR.ansys.HfssEMSetup'>)
INFO 10:55PM [analyze]: Analyzing setup Setup
10:55PM 51s INFO [get_f_convergence]: Saved convergences to
C:\Users\Bartu\Desktop\qiskit-metal\tutorials\4 Analysis\A. Core - EM and
quantization\hfss_eig_f_convergence.csv
```

```
[12]: re.Mode(1)) [g]
Pass []
1          40.055249
2          4.850279
3          5.816798
4          6.039345
5          6.151084
6          6.222545
```

Verify that the Electro(magnetic) fields look realistic.

```
[13]: eig_qb.sim.plot_fields('main')    # TODO:::: Ez, normal component.....
      ↪decide which field typically on the qbit, or on the crossing between
      ↪meanders
eig_qb.sim.save_screenshot()
```

```
INFO 12:00AM [get_setup]:      Opened setup `Setup`  (<class
'pyEPR.ansys.HfssEMSetup'>)
```



```
[13]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/A.u  
→Core -  
EM and quantization/ansys.png')
```

(optional) clear the renderer by removing the fields

```
[14]: eig_qb.sim.clear_fields()
```

3.11.0.3 EPR Analysis

3.11.0.3.1 Setup

Identify the non-linear (Josephson) junctions in the model. You will need to list the junctions in the epr setup.

In this case there's only one junction, namely 'jj'. Let's see what we need to change in the default setup.

```
[15]: eig_qb.setup
```

```
[15]: {'junctions': {'jj': {'Lj_variable': 'Lj',  
'Cj_variable': 'Cj',  
'rect': '',  
'line': ''}},  
'dissipatives': {'dielectrics_bulk': ['main']},  
'cos_trunc': 8,  
'fock_trunc': 7,  
'sweep_variable': 'Lj'}
```

The name of the Lj_variable and Cj_variable match with our model. However it is missing the names of the shapes that identify the junction (rect and line). Look for those in the renderer and find the name. Then let's change the name (See below).

```
[16]: eig_qb.setup.junctions.jj.rect = 'JJ_rect_Lj_Q1_rect_jj'  
eig_qb.setup.junctions.jj.line = 'JJ_Lj_Q1_rect_jj_'  
eig_qb.setup
```

```
[16]: {'junctions': {'jj': {'Lj_variable': 'Lj',  
'Cj_variable': 'Cj',  
'rect': 'JJ_rect_Lj_Q1_rect_jj',  
'line': 'JJ_Lj_Q1_rect_jj_'}},  
'dissipatives': {'dielectrics_bulk': ['main']},  
'cos_trunc': 8,  
'fock_trunc': 7,
```

```
'sweep_variable': 'Lj'}
```

We will now run epr as a single step. On screen you will observe various information in this order:

- * stored energy = Electric and magnetic energy stored in the substrate and the system as a whole.
- * EPR analysis results for all modes/variations.
- * Spectrum analysis.
- * Hamiltonian report.

[17]: eig_qb.run_epr()

```
#### equivalent individual calls
# s = self.setup
# self.epr_start()
# eig_qb.get_stored_energy()
# eig_qb.run_analysis()
# eig_qb.spectrum_analysis(s.cos_trunc, s.fock_trunc)
# eig_qb.report_hamiltonian(s.sup_variable)
```

Design "Qbit_hfss" info:

```
# eigenmodes    1
# variations    1
```

Design "Qbit_hfss" info:

```
# eigenmodes    1
# variations    1
```

```
energy_elec_all      = 7.4914236101879e-24
energy_elec_substrate = 6.88978482985149e-24
EPR of substrate = 92.0%
```

```
energy_mag      = 3.89880173688021e-26
energy_mag % of energy_elec_all = 0.5%
```

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for empty

Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

```
options=pd.Series(get_instance_vars(self.options)),
```

Variation 0 [1/1]

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1096: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future

version. Specify a dtype explicitly to silence this warning.

```
Ljs = pd.Series({})
```

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1097: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

```
Cjs = pd.Series({})
```

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

```
_Om = pd.Series({})
```

Mode 0 at 6.22 GHz [1/1]
Calculating _magnetic,_electric

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

```
Sj = pd.Series({})
```

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

```
Qp = pd.Series({})
```

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The series.append method is deprecated and will be removed from pandas in a future

version. Use pandas.concat instead.

```
sol = sol.append(self.get_Qdielectric(
```

(_E-_H)/_E	_E	_H
99.5%	3.746e-24	1.949e-26

Calculating junction energy participation ration (EPR)
method='line_voltage'. First estimates:
junction EPR p_0j sign s_0j (p_capacitive)

```

Energy fraction (Lj over Lj&Cj)= 96.75%
jj           0.904052 (+)      0.0304027
(U_tot_cap-U_tot_ind)/mean=6.25%
Calculating Qdielectric_main for mode 0 (0/0)
p_dielectric_main_0 = 0.9196896595837651

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for
    ↪empty
Series will be 'object' instead of 'float64' in a future version. Specify a
dtype explicitly to silence this warning.
    options=pd.Series(get_instance_vars(self.options)),

WARNING 12:03AM [__init__]: <p>Error: <class 'IndexError'></p>
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_quantum_analysis.py:715: FutureWarning: Support for
    ↪multi-
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be
    ↪removed in
a future version. Convert to a numpy array before indexing instead.
    result['Q_coupling'] = self.Qm_coupling[vary][self.
    ↪Qm_coupling[vary]
.columns[junctions]][modes]#TODO change the columns to junctions

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_quantum_analysis.py:720: FutureWarning: Support for
    ↪multi-
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be
    ↪removed in
a future version. Convert to a numpy array before indexing instead.
    result['Qs'] =
self.Qs[vary][self.PM[vary].columns[junctions]][modes] #TODO
    ↪change
the columns to junctions

```

ANALYSIS DONE. Data saved to:

C:\data-pyEPR\Project1\Qbit_hfss\2022-04-26 00-03-33.npz

Differences in variations:

```
.....  
↪ ..
```

Variation 0

```
Starting the diagonalization
Finished the diagonalization
Pm_norm=
modes
0    1.133644
dtype: float64

Pm_norm idx =
    jj
0  True
*** P (participation matrix, not normlz.)
    jj
0  0.877377

*** S (sign-bit matrix)
s_jj
0    1
*** P (participation matrix, normalized.)
    0.99

*** Chi matrix 01 PT (MHz)
Diag is anharmonicity, off diag is full cross-Kerr.
    322

*** Chi matrix ND (MHz)
    361

*** Frequencies 01 PT (MHz)
0    5900.326844
dtype: float64

*** Frequencies ND (MHz)
0    5881.45938
dtype: float64

*** Q_coupling
Empty DataFrame
Columns: []
Index: [0]
```

3.11.0.3.2 Mode frequencies (MHz)

Numerical diagonalization

Lj	11
0	5881.46

3.11.0.3.3 Kerr Non-linear coefficient table (MHz)

Numerical diagonalization

Lj	0
11 0	361.36

3.12 2. Analyze the CPW resonator by itself

3.12.0.1 Update the design in Metal

Connect the transmon to a CPW. The other end of the CPW connects to an open to ground termination.

```
[18]: from qiskit_metal qlibrary terminations open_to_ground import OpenToGround
from qiskit_metal qlibrary tlines meandered import RouteMeander
otg = OpenToGround(design, 'open_to_ground', options=dict(pos_x='1.75mm',
    pos_y='0um', orientation='0'))
RouteMeander(design, 'readout', Dict(
    total_length='6 mm',
    hfss_wire_bonds = True,
    fillet='90 um',
    lead = dict(start_straight='100um'),
    pin_inputs=Dict(
        start_pin=Dict(component='Q1', pin='readout'),
        end_pin=Dict(component='open_to_ground', pin='open'))))

gui.rebuild()
gui.autoscale()
```

3.12.0.2 Finite Element Eigenmode Analysis

3.12.0.2.1 Setup

Create a separate analysis object, dedicated to the readout. This allows to retain the Qubit session active, in case we will later need to tweak the design and repeat the simulation. When different renderers are available you could even consider using different more appropriate ones for each simulation steps of this notebook, but for now we will be using the same one.

```
[19]: eig_rd = EPRanalysis(design, "hfss")
```

For the resonator analysis we will use the default setup. You can feel free to edit it the same way we did in section 1.

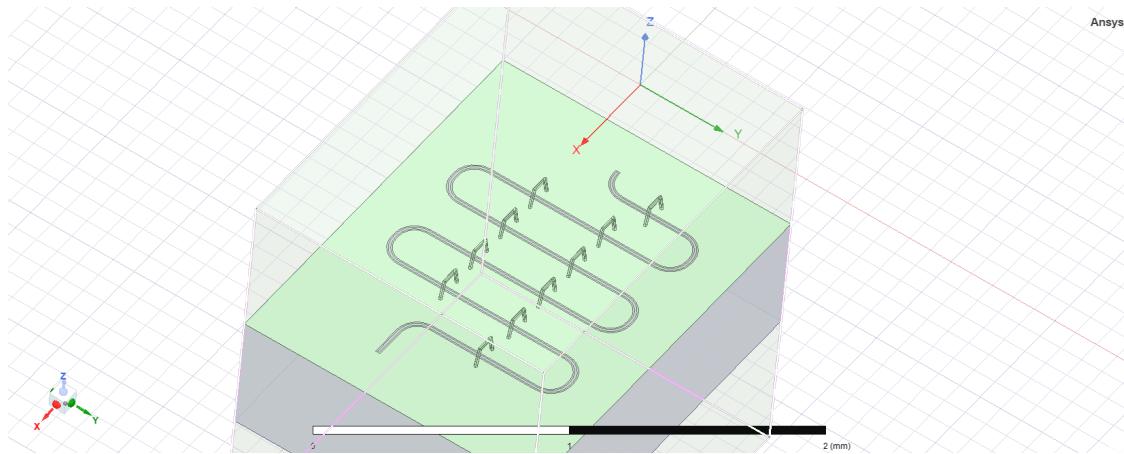
3.12.0.2.2 Execute simulation and verify convergence and EM field

Analyze the readout in isolation. Select the readout and terminate it with an open on both ends. Note that we are selecting for this analysis both the `readout` component and the `open_to_ground` component. The `open_to_ground` component might feel redundant because we are specifying in that open in the `open_terminations`, and the end converging result is indeed the same. However the `open_to_ground` appears to help the system to converge faster, so we keep it in there.

```
[20]: eig_rd.sim.run(name="Readout",
                     components=['readout', 'open_to_ground'],
                     open_terminations=[('readout', 'start'), ('readout', 'end')])
eig_rd.sim.plot_convergences()
```

```
INFO 12:13AM [connect_design]: Opened active design
      Design: Readout_hfss [Solution type: Eigenmode]
WARNING 12:13AM [connect_setup]: No design setup detected.
WARNING 12:13AM [connect_setup]: Creating eigenmode default setup.
INFO 12:13AM [get_setup]: Opened setup `Setup` (<class
'pyEPR.anys.HfssEMSetup'>)
INFO 12:13AM [get_setup]: Opened setup `Setup` (<class
'pyEPR.anys.HfssEMSetup'>)
INFO 12:13AM [analyze]: Analyzing setup Setup
12:14AM 52s INFO [get_f_convergence]: Saved convergences to
C:\Users\Bartu\Desktop\qiskit-metal\tutorials\4 Analysis\A. Core - EM and
quantization\hfss_eig_f_convergence.csv
```

```
[21]: eig_rd.sim.save_screenshot() # optional
```



[21]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/A.
 ↪Core -
 EM and quantization/ansys.png')

Recover eigenmode frequencies for each variation.

[22]: eig_rd.get_frequencies()

```
Design "Readout_hfss" info:  

    # eigenmodes      1  

    # variations      1  

Design "Readout_hfss" info:  

    # eigenmodes      1  

    # variations      1
```

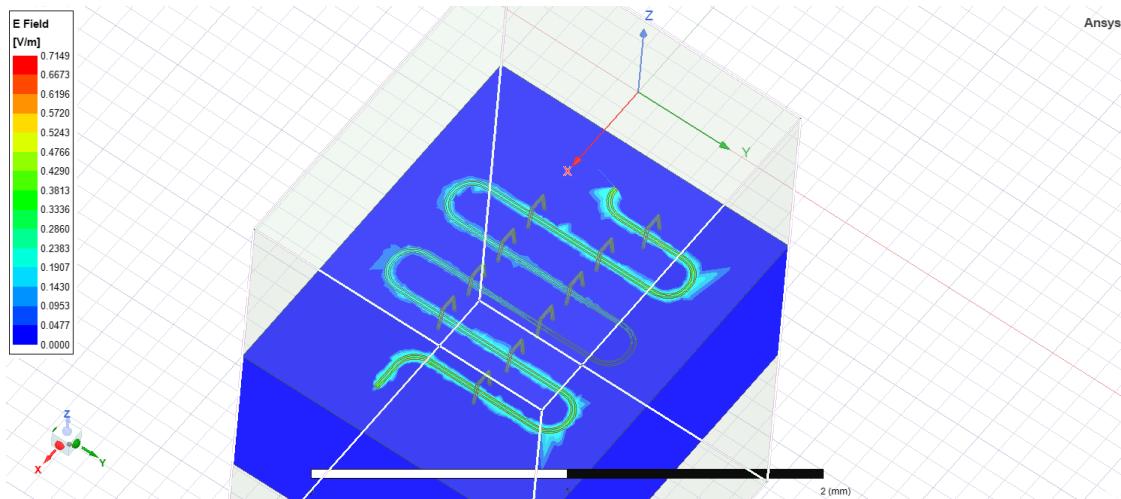
	Freq. (GHz)	Quality Factor
variation mode		
0	0	9.696963 inf

Display the Ansys modeler window and plot the E-field on the chip's surface.

[23]: eig_rd.sim.plot_fields('main')
 eig_rd.sim.save_screenshot()

```
INFO 12:20AM [get_setup]:      Opened setup `Setup`  (<class  

'pyEPR.ansys.HfssEMSetup'>)
```



```
[23]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/A.  
↳Core -  
EM and quantization/ansys.png')
```

3.12.0.2.3 Refine

If convergence is not complete, or the EM field is unclear, update the number of passes and re-run the flow (below repeated for convenience)

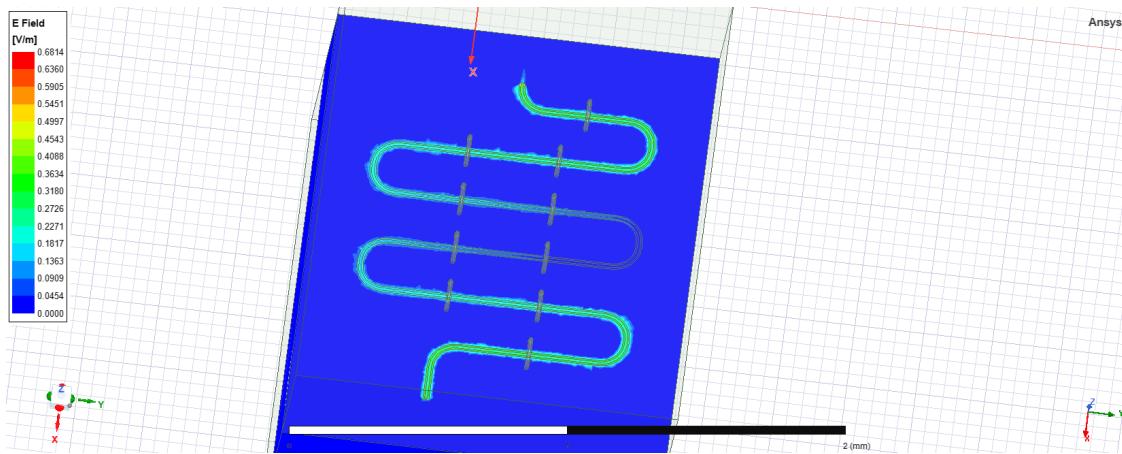
```
[24]: eig_rd.sim.setup.max_passes = 15    # update single setting  
eig_rd.sim.run()  
eig_rd.sim.plot_convergences()
```

```
INFO 12:21AM [get_setup]:      Opened setup `Setup` (<class  
'pyEPR.ansys.HfssEMSetup'>)  
INFO 12:21AM [analyze]: Analyzing setup Setup  
12:23AM 42s INFO [get_f_convergence]: Saved convergences to  
C:\Users\Bartu\Desktop\qiskit-metal\tutorials\4 Analysis\A. Core - EM and  
quantization\hfss_eig_f_convergence.csv
```

Display the Ansys modeler window again and plot the E-field on the chip's surface with this updated number of passes. Note that the bright areas have become much smoother compared to the previous image, indicating better convergence.

```
[25]: eig_rd.sim.plot_fields('main')  
eig_rd.sim.save_screenshot()
```

```
INFO 12:23AM [get_setup]:      Opened setup `Setup` (<class  
'pyEPR.ansys.HfssEMSetup'>)
```



[25] : WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/A.
 ↪Core -
 EM and quantization/ansys.png')

3.12.0.3 EPR Analysis

Find the electric and magnetic energy stored in the readout system.

[26] : eig_rd.run_epr(no_junctions = True)

```
Design "Readout_hfss" info:
    # eigenmodes      1
    # variations      1
Design "Readout_hfss" info:
    # eigenmodes      1
    # variations      1

    energy_elec_all      = 2.88142035747661e-24
    energy_elec_substrate = 2.63396356206577e-24
    EPR of substrate = 91.4%

    energy_mag      = 2.88147272532409e-24
    energy_mag % of energy_elec_all = 100.0%
```

3.13 3. Analyze the combined transmon + CPW resonator system

3.13.0.1 Finite Element Eigenmode Analysis

3.13.0.1.1 Setup

Create a separate analysis object for the combined qbit+readout.

```
[27]: eig_qres = EPRanalysis(design, "hfss")
```

For the resonator analysis we look for 2 eigenmodes - one with stronger fields near the transmon, the other with stronger fields near the resonator. Therefore let's update the setup accordingly.

```
[28]: eig_qres.sim.setup.n_modes = 2  
eig_qres.sim.setup
```

```
[28]: {'name': 'Setup',  
       'reuse_selected_design': True,  
       'reuse_setup': True,  
       'min_freq_ghz': 1,  
       'n_modes': 2,  
       'max_delta_f': 0.5,  
       'max_passes': 10,  
       'min_passes': 1,  
       'min_converged': 1,  
       'pct_refinement': 30,  
       'basis_order': 1,  
       'vars': {'Lj': '10 nH', 'Cj': '0 fF'}}
```

3.13.0.1.2 Execute simulation and verify convergence and EM field

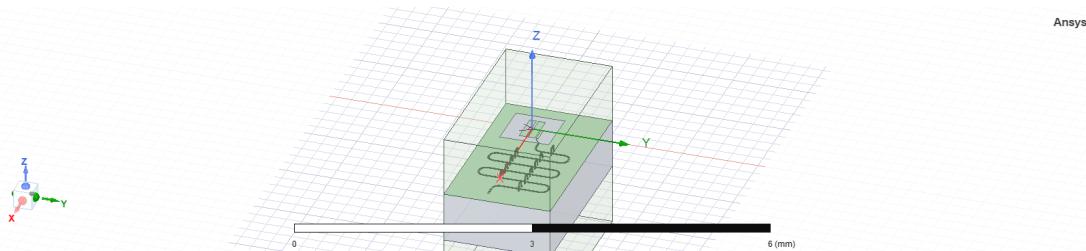
Analyze the qubit+readout. Select the qubit and the readout, then finalize with open termination on the other pins.

```
[29]: eig_qres.sim.run(name="TransmonResonator",  
                      components=['Q1', 'readout', 'open_to_ground'],  
                      open_terminations=[('readout', 'end')])  
eig_qres.sim.plot_convergences()
```

```
INFO 12:24AM [connect_design]: Opened active design  
Design: TransmonResonator_hfss [Solution type: Eigenmode]  
WARNING 12:24AM [connect_setup]: No design setup detected.  
WARNING 12:24AM [connect_setup]: Creating eigenmode default setup.  
INFO 12:24AM [get_setup]: Opened setup `Setup` (<class  
'pyEPR.ansys.HfssEMSetup'>)
```

```
INFO 12:25AM [get_setup]:      Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
INFO 12:25AM [analyze]: Analyzing setup Setup
12:26AM 34s INFO [get_f_convergence]: Saved convergences to
C:\Users\Bartu\Desktop\qiskit-metal\tutorials\4 Analysis\A. Core - EM and
quantization\hfss_eig_f_convergence.csv
```

```
[30]: eig_qres.sim.save_screenshot() # optional
```

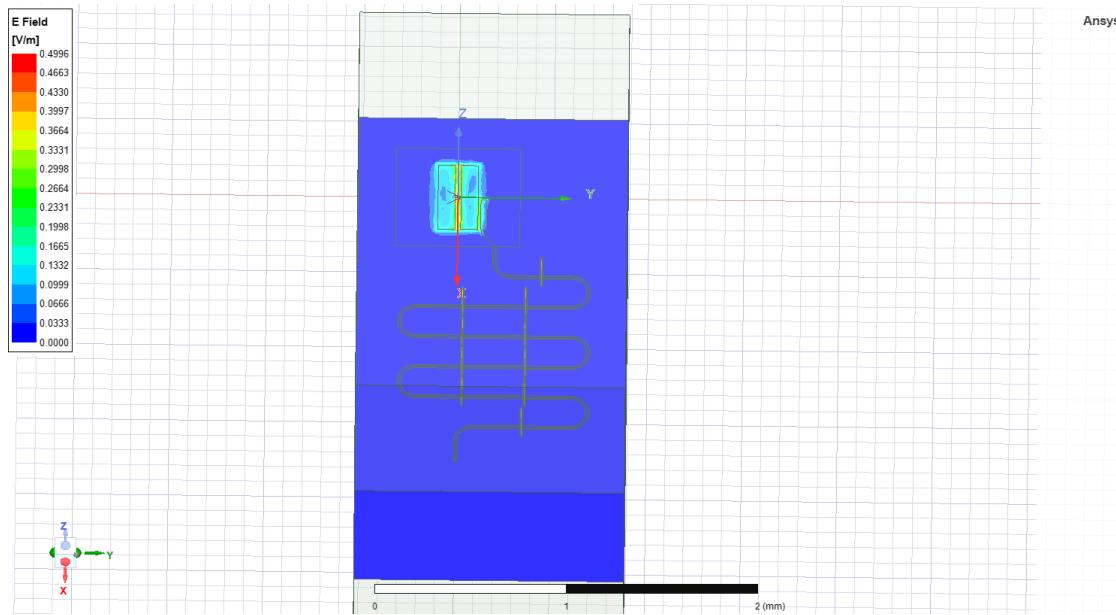


```
[30]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/A. Core -
          EM and quantization/ansys.png')
```

Display the Ansys modeler window again and plot the E-field on the chip's surface. you can select which of the two modes to visualize.

```
[31]: eig_qres.sim.plot_fields('main', eigenmode=1)
eig_qres.sim.save_screenshot()
```

```
INFO 12:27AM [get_setup]:      Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
```



[31]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/A.
 ↪Core -
 EM and quantization/ansys.png')

3.13.0.2 EPR Analysis

Similarly to section 1, we need to pass to the renderer the names of the shapes that identify the junction (rect and line). These should be the same as in section 1, or you can look again for those in the renderer.

```
[32]: eig_qres.setup.junctions.jj.rect = 'JJ_rect_Lj_Q1_rect_jj'  

eig_qres.setup.junctions.jj.line = 'JJ_Lj_Q1_rect_jj_'  

eig_qres.setup
```

```
[32]: {'junctions': {'jj': {'Lj_variable': 'Lj',  

'Cj_variable': 'Cj',  

'rect': 'JJ_rect_Lj_Q1_rect_jj',  

'line': 'JJ_Lj_Q1_rect_jj_'}},  

'dissipatives': {'dielectrics_bulk': ['main']},  

'cos_trunc': 8,  

'fock_trunc': 7,  

'sweep_variable': 'Lj'}
```

We will now run epr as a single step. On screen you will observe various information in this order:

- * stored energy = Electric and magnetic energy stored in the substrate and the system as a whole.
- * EPR analysis results for all modes/variations.
- * Spectrum analysis.

Hamiltonian report.

```
[33]: eig_qres.run_epr()
```

```
Design "TransmonResonator_hfss" info:  
    # eigenmodes      2  
    # variations      1  
Design "TransmonResonator_hfss" info:  
    # eigenmodes      2  
    # variations      1
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-  
packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for  
empty
```

```
Series will be 'object' instead of 'float64' in a future version. Specify a  
dtype explicitly to silence this warning.  
options=pd.Series(get_instance_vars(self.options)),
```

```
energy_elec_all      = 5.90681516286258e-24  
energy_elec_substrate = 5.4184308659889e-24  
EPR of substrate = 91.7%  
  
energy_mag      = 5.14784217340854e-26  
energy_mag % of energy_elec_all = 0.9%
```

Variation 0 [1/1]

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-  
packages\pyEPR\core_distributed_analysis.py:1096: FutureWarning: The default  
dtype for empty Series will be 'object' instead of 'float64' in a future  
version. Specify a dtype explicitly to silence this warning.
```

```
Ljs = pd.Series({})
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-  
packages\pyEPR\core_distributed_analysis.py:1097: FutureWarning: The default  
dtype for empty Series will be 'object' instead of 'float64' in a future  
version. Specify a dtype explicitly to silence this warning.
```

```
Cjs = pd.Series({})
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-  
packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default  
dtype for empty Series will be 'object' instead of 'float64' in a future  
version. Specify a dtype explicitly to silence this warning.
```

```

_0m = pd.Series({})

Mode 0 at 6.18 GHz [1/2]
Calculating _magnetic,_electric

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
Sj = pd.Series({})

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
Qp = pd.Series({})

(_E-_H)/_E      _E      _H
99.1%  2.953e-24  2.574e-26

Calculating junction energy participation ration (EPR)
method='line_voltage'. First estimates:
junction      EPR p_0j    sign s_0j    (p_capacitive)
Energy fraction (Lj over Lj&Cj)= 97.08%
jj            0.99076 (+)      0.0298401
(U_tot_cap-U_tot_ind)/mean=1.50%
Calculating Qdielectric_main for mode 0 (0/1)

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The
series.append method is deprecated and will be removed from pandas in a
future
version. Use pandas.concat instead.
sol = sol.append(self.get_Qdielectric()

p_dielectric_main_0 = 0.9173185069435967

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
_0m = pd.Series({})

```

```

Mode 1 at 9.29 GHz [2/2]
Calculating _magnetic,_electric

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.

Sj = pd.Series({})

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.

Qp = pd.Series({})

(_E-_H)/_E      _E      _H
0.2%  2.183e-24  2.177e-24

Calculating junction energy participation ration (EPR)
method='line_voltage'. First estimates:
junction      EPR p_1j    sign s_1j    (p_capacitive)
Energy fraction (Lj over Lj&Cj)= 93.61%
jj            0.00310906 (+)        0.000212083
(U_tot_cap-U_tot_ind)/mean=-0.03%
Calculating Qdielectric_main for mode 1 (1/1)

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The
series.append method is deprecated and will be removed from pandas in a
future
version. Use pandas.concat instead.

sol = sol.append(self.get_Qdielectric()

p_dielectric_main_1 = 0.9129942054088999

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for
empty
Series will be 'object' instead of 'float64' in a future version. Specify a
dtype explicitly to silence this warning.

options=pd.Series(get_instance_vars(self.options)),

WARNING 12:28AM [__init__]: <p>Error: <class 'IndexError'></p>
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-

```

```
packages\pyEPR\core_quantum_analysis.py:715: FutureWarning: Support for
    ↵multi-
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be
    ↵removed in
a future version. Convert to a numpy array before indexing instead.
    result['Q_coupling'] = self.Qm_coupling[varyation][self.
    ↵Qm_coupling[varyation]
.columns[junctions]][modes]#TODO change the columns to junctions

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_quantum_analysis.py:720: FutureWarning: Support for
    ↵multi-
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be
    ↵removed in
a future version. Convert to a numpy array before indexing instead.
    result['Qs'] =
self.Qs[varyation][self.PM[varyation].columns[junctions]][modes] #TODO
    ↵change
the columns to junctions
```

ANALYSIS DONE. Data saved to:

C:\data-pyEPR\Project1\TransmonResonator_hfss\2022-04-26 00-27-58.npz

Differences in variations:

.....
↳ ..

Variation 0

```
Starting the diagonalization
Finished the diagonalization
Pm_norm=
modes
0      1.030517
1      0.912917
dtype: float64
```

```
Pm_norm idx =
    jj
```

```
0  True
1  False
*** P (participation matrix, not normlz.)
      jj
0  0.962052
1  0.003108

*** S (sign-bit matrix)
      s_jj
0    1
1    1
*** P (participation matrix, normalized.)
      0.99
      0.0031

*** Chi matrix 01 PT (MHz)
Diag is anharmonicity, off diag is full cross-Kerr.
      287    2.71
      2.71  0.00638

*** Chi matrix ND (MHz)
      319    2.31
      2.31  0.00469

*** Frequencies 01 PT (MHz)
0    5888.131422
1    9293.519740
dtype: float64

*** Frequencies ND (MHz)
0    5872.958894
1    9293.572555
dtype: float64

*** Q_coupling
Empty DataFrame
Columns: []
Index: [0, 1]
```

3.13.0.2.1 Mode frequencies (MHz)

Numerical diagonalization

```
Lj      10
0    5872.96
1    9293.57
```

3.13.0.2.2 Kerr Non-linear coefficient table (MHz)

Numerical diagonalization

Lj	0	1
10 0	318.98	2.31e+00
1	2.31	4.69e-03

Once you are sure you are done with the qubit analysis, please explicitly release the Ansys session to allow for a smooth close of the external tool.

[34] : eig_qb.sim.close()

[35] : eig_rd.sim.close()

[36] : eig_qres.sim.close()

3.14 4. Analyze a coupled 2-transmon system

3.14.0.1 Create the design

This is a different system than the one analyzed in sections 1,2,3. Therefore, let's start by deleting the design currently in the Qiskit Metal GUI (if any).

[37] : design.delete_all_components()

Next, we create the TwoTransmon design, consisting of 2 transmons connected by a short coupler.

```
[38]: from qiskit_metal qlibrary qubits transmon_pocket import TransmonPocket
from qiskit_metal qlibrary tlines straight_path import RouteStraight

q1 = TransmonPocket(design, 'Q1', options = dict(
    pad_width = '425 um',
    pocket_height = '650um',
    connection_pads=dict(
        readout = dict(loc_W=+1, loc_H=+1, pad_width='200um')
    )))
q2 = TransmonPocket(design, 'Q2', options = dict(
    pos_x = '1.0 mm',
    pad_width = '425 um',
    pocket_height = '650um',
    connection_pads=dict(
        readout = dict(loc_W=-1, loc_H=+1, pad_width='200um')
    ))
```

```
)))
coupler = RouteStraight(design, 'coupler', Dict(hfss_wire_bonds = True,
pin_inputs=Dict(
    start_pin=Dict(component='Q1', pin='readout'),
    end_pin=Dict(component='Q2', pin='readout')), ))
gui.rebuild()
gui.autoscale()
```

Let's observe the current table describing the junctions in the qiskit metal design

[39]: `design.qgeometry.tables['junction']`

```
[39]:   component      name                      geometry
       ↳layer \
0        4 rect_jj  LINESTRING (0.00000 -0.01500, 0.00000 0.01500)  ↳
       ↳1
1        5 rect_jj  LINESTRING (1.00000 -0.01500, 1.00000 0.01500)  ↳
       ↳1

      subtract  helper  chip  width hfss_inductance  hfss_capacitance \
0     False    False  main    0.02           10nH            0
1     False    False  main    0.02           10nH            0

      hfss_resistance  hfss_mesh_kw_jj  q3d_inductance  q3d_capacitance \
0                  0          0.000007        10nH            0
1                  0          0.000007        10nH            0

      q3d_resistance  q3d_mesh_kw_jj      gds_cell_name
0                  0          0.000007  my_other_junction
1                  0          0.000007  my_other_junction
```

You can observe in the table above that every junction has been assigned a default inductance, capacitance and resistance values, based on the originating component class `default_options`. In this example we intend to replace those values with a variable name, which will later be set directly in the renderer. Therefore, let's proceed with updating these values in the qubit instances, and then propagate the update to the table with a `rebuild()`. After executing the cell below, you can observe the change by re-executing the cell above.

[40]: `# TODO: fold this inside either an analysis class method, or inside the
 ↳analysis class setup`

```
qcamps = design.components # short handle (alias)
```

```

qcomps['Q1'].options['hfss_inductance'] = 'Lj1'
qcomps['Q1'].options['hfss_capacitance'] = 'Cj1'
qcomps['Q2'].options['hfss_inductance'] = 'Lj2'
qcomps['Q2'].options['hfss_capacitance'] = 'Cj2'
gui.rebuild() # line needed to propagate the updates from the qubit
    ↪ instance into the junction design table
gui.autoscale()

```

3.14.0.2 Finite Element Eigenmode Analysis

3.14.0.2.1 Setup

Let's start the analysis by creating the appropriate analysis object.

```
[41]: from qiskit_metal.analyses.quantization import EPRanalysis
eig_2qb = EPRanalysis(design, "hfss")
```

Now let us update the setup of this analysis to reflect what we plan to do: * define the variables that we have assigned to the inductance and capacitance of the junctions; * increase accuracy of the convergence; * observe the eigenmode corresponding to both qubits.

```
[42]: eig_2qb.sim.setup.max_passes = 15
eig_2qb.sim.setup.max_delta_f = 0.05
eig_2qb.sim.setup.n_modes = 2
eig_2qb.sim.setup.vars = Dict(Lj1= '13 nH', Cj1= '0 fF',
                               Lj2= '9 nH', Cj2= '0 fF')
eig_2qb.sim.setup
```

```
[42]: {'name': 'Setup',
       'reuse_selected_design': True,
       'reuse_setup': True,
       'min_freq_ghz': 1,
       'n_modes': 2,
       'max_delta_f': 0.05,
       'max_passes': 15,
       'min_passes': 1,
       'min_converged': 1,
       'pct_refinement': 30,
       'basis_order': 1,
       'vars': {'Lj1': '13 nH', 'Cj1': '0 fF', 'Lj2': '9 nH', 'Cj2': '0 fF'}}
```

By default, the analysis will be done on all components that we will list in the `run_sim()` method, but the analysis needs to know how much of the ground plane around the qubit to consider. One could use the declared chip dimension by passing the parameter `bux_plus_buffer = False` to the `run_sim()` method. However, its default (when said parameter is omitted) is to consider the ground plane to be as big as the minimum enclosing

rectangle plus a set buffer. The default buffer value is 200um, while in the cell below we will increase as an example that buffer to 500um.

```
[43]: # TODO: fold this inside either an analysis class method, or inside the
      →analysis class setup
```

```
eig_2qb.sim.renderer.options['x_buffer_width_mm'] = 0.5
eig_2qb.sim.renderer.options['y_buffer_width_mm'] = 0.5
eig_2qb.sim.renderer.options
```

```
[43]: {'Lj': '10nH',
       'Cj': 0,
       '_Rj': 0,
       'max_mesh_length_jj': '7um',
       'project_path': None,
       'project_name': None,
       'design_name': None,
       'x_buffer_width_mm': 0.5,
       'y_buffer_width_mm': 0.5,
       'wb_threshold': '400um',
       'wb_offset': '0um',
       'wb_size': 5,
       'plot_ansys_fields_options': {'name': 'NAME:Mag_E1',
                                     'UserSpecifyName': '0',
                                     'UserSpecifyFolder': '0',
                                     'QuantityName': 'Mag_E',
                                     'PlotFolder': 'E Field',
                                     'StreamlinePlot': 'False',
                                     'AdjacentSidePlot': 'False',
                                     'FullModelPlot': 'False',
                                     'IntrinsicVar': "Phase='0deg'",
                                     'PlotGeomInfo_0': '1',
                                     'PlotGeomInfo_1': 'Surface',
                                     'PlotGeomInfo_2': 'FacesList',
                                     'PlotGeomInfo_3': '1'}}}
```

Let's finally run the cap extraction simulation and observe the convergence.

```
[44]: eig_2qb.sim.run(name="TwoTransmons",
                      components=['coupler', 'Q1', 'Q2'])
```

```
INFO 12:29AM [connect_project]: Connecting to Ansys Desktop API...
INFO 12:29AM [load_ansys_project]:     Opened Ansys App
INFO 12:29AM [load_ansys_project]:     Opened Ansys Desktop v2022.1.0
INFO 12:29AM [load_ansys_project]:     Opened Ansys Project
```

```

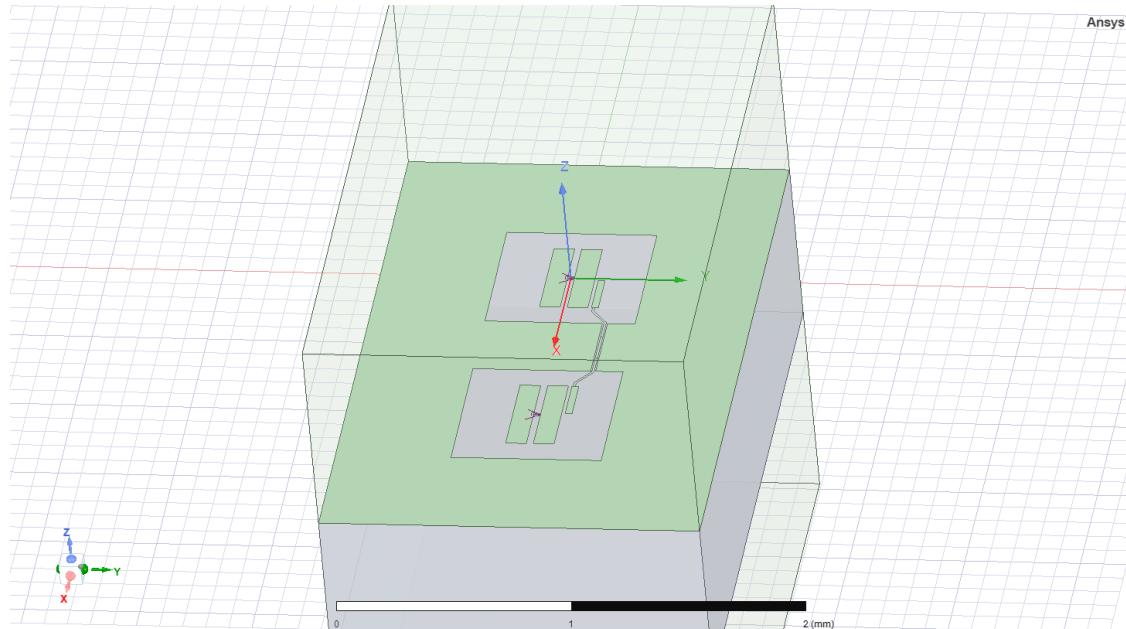
Folder:      C:/Users/Bartu/Documents/Ansoft/
Project:    Project1
INFO 12:29AM [connect_design]:  Opened active design
Design:      Qbit_hfss [Solution type: Eigenmode]
INFO 12:29AM [get_setup]:      Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
INFO 12:29AM [connect]:        Connected to project "Project1" and design
"Qbit_hfss"

INFO 12:29AM [connect_design]:  Opened active design
Design:      TwoTransmons_hfss [Solution type: Eigenmode]
WARNING 12:29AM [connect_setup]: No design setup detected.
WARNING 12:29AM [connect_setup]: Creating eigenmode default setup.
INFO 12:29AM [get_setup]:      Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
INFO 12:29AM [get_setup]:      Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
INFO 12:29AM [analyze]: Analyzing setup Setup
12:33AM 30s INFO [get_f_convergence]: Saved convergences to
C:\Users\Bartu\Desktop\qiskit-metal\tutorials\4 Analysis\A. Core - EM and
quantization\hfss_eig_f_convergence.csv

```

[45]: `eig_2qb.sim.plot_convergences()`

[46]: `eig_2qb.sim.save_screenshot() # optional`

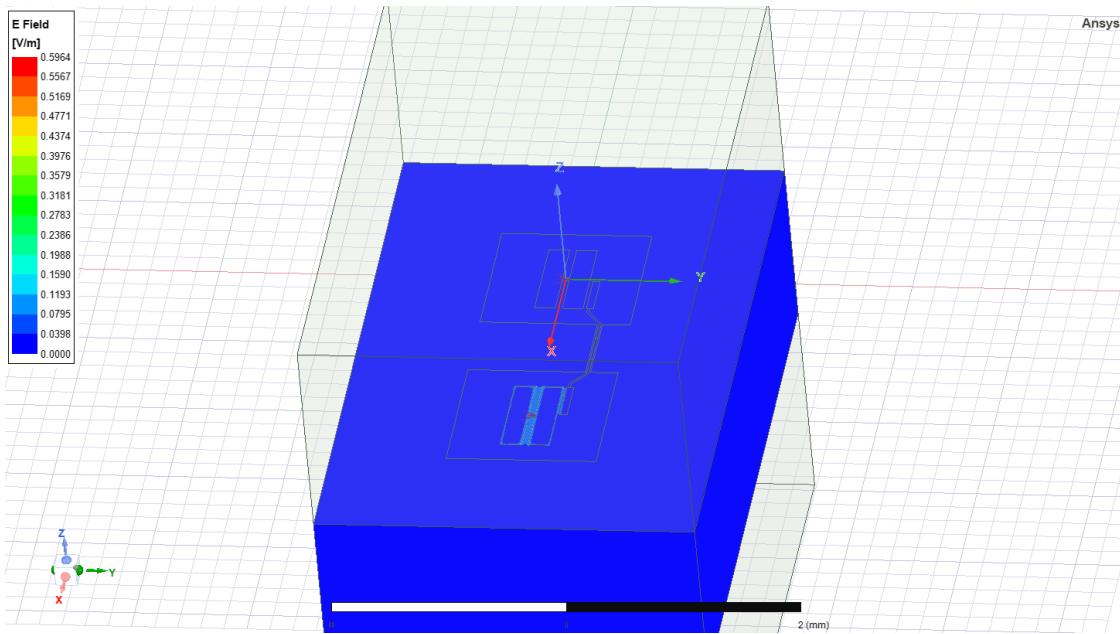


[46]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/A.
 ↪Core -
 EM and quantization/ansys.png')

Display the Ansys modeler window again and plot the E-field on the chip's surface. Since we have analyzed 2 modes, you will need to select which mode to visualize. The default is mode 1, but the mode can inclusively be any integer between 1 and `setup.n_modes`.

[47]: `eig_2qb.sim.plot_fields('main', eigenmode=2)`
`eig_2qb.sim.save_screenshot()`

INFO 12:38AM [get_setup]: Opened setup `Setup` (<class
`'pyEPR.ansys.HfssEMSetup'`>)



[47]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/A.
 ↪Core -
 EM and quantization/ansys.png')

3.14.0.3 EPR Analysis

3.14.0.3.1 Setup

Identify the non-linear (Josephson) junctions in the model. in this case there are 2 junctions, which we will refer to as `jj1` and `jj2`. Also define the dissipative reference shapes. Remove the default junction and create the two.

```
[48]: del eig_2qb.setup.junctions['jj']
```

```
[49]: eig_2qb.setup.junctions.jj1 = Dict(rect='JJ_rect_Lj_Q1_rect_jj',  
    ↪line='JJ_Lj_Q1_rect_jj_ ',  
        ↪Lj_variable='Lj1', Cj_variable='Cj1')  
eig_2qb.setup.junctions.jj2 = Dict(rect='JJ_rect_Lj_Q2_rect_jj',  
    ↪line='JJ_Lj_Q2_rect_jj_ ',  
        ↪Lj_variable='Lj2', Cj_variable='Cj2')  
eig_2qb.setup.sweep_variable = 'Lj1'  
eig_2qb.setup
```

```
[49]: {'junctions': {'jj1': {'rect': 'JJ_rect_Lj_Q1_rect_jj',  
    'line': 'JJ_Lj_Q1_rect_jj_ ',  
    'Lj_variable': 'Lj1',  
    'Cj_variable': 'Cj1'},  
    'jj2': {'rect': 'JJ_rect_Lj_Q2_rect_jj',  
    'line': 'JJ_Lj_Q2_rect_jj_ ',  
    'Lj_variable': 'Lj2',  
    'Cj_variable': 'Cj2'}},  
    'dissipatives': {'dielectrics_bulk': ['main']},  
    'cos_trunc': 8,  
    'fock_trunc': 7,  
    'sweep_variable': 'Lj1'}
```

Find the electric and magnetic energy stored in the substrate and the system as a whole.

```
[50]: eig_2qb.run_epr()
```

```
Design "TwoTransmons_hfss" info:  
    # eigenmodes      2  
    # variations      1  
Design "TwoTransmons_hfss" info:  
    # eigenmodes      2  
    # variations      1
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-  
packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for  
    ↪empty  
Series will be 'object' instead of 'float64' in a future version. Specify a  
dtype explicitly to silence this warning.  
    options=pd.Series(get_instance_vars(self.options)),
```

```
energy_elec_all      = 2.90307377390687e-25  
energy_elec_substrate = 2.67601671391633e-25
```

EPR of substrate = 92.2%

energy_mag = 1.71590650735572e-27
 energy_mag % of energy_elec_all = 0.6%

Variation 0 [1/1]

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1096: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

Ljs = pd.Series({})

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1097: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

Cjs = pd.Series({})

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

_Om = pd.Series({})

Mode 0 at 5.62 GHz [1/2]

Calculating _magnetic,_electric

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

Sj = pd.Series({})

(_E-_H)/_E	_E	_H
99.6%	3.269e-25	1.323e-27

Calculating junction energy participation ration (EPR)
 method='line_voltage'. First estimates:
 junction EPR p_0j sign s_0j (p_capacitive)
 Energy fraction (Lj over Lj&Cj)= 96.85%
 jj1 0.995338 (+) 0.0323214

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

```
Qp = pd.Series({})
```

	Energy fraction (Lj over Lj&Cj)= 97.80%
jj2	0.000377757 (+) 8.49242e-06
	(U_tot_cap-U_tot_ind)/mean=1.60%

Calculating Qdielectric_main for mode 0 (0/1)

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The series.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

```
sol = sol.append(self.get_Qdielectric(
```

p_dielectric_main_0 = 0.921603405436269

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

```
_Om = pd.Series({})
```

Mode 1 at 6.76 GHz [2/2]
Calculating _magnetic,_electric

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

```
Sj = pd.Series({})
```

(_E-_H)/_E	_E	_H
99.4%	1.452e-25	8.58e-28

Calculating junction energy participation ration (EPR)
method='line_voltage'. First estimates:
junction EPR p_1j sign s_1j (p_capacitive)
Energy fraction (Lj over Lj&Cj)= 95.52%
jj1 0.000382152 (+) 1.79374e-05

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.
```

```
Qp = pd.Series({})
```

```
Energy fraction (Lj over Lj&Cj)= 96.85%
jj2          0.99282 (+)      0.0322622
(U_tot_cap-U_tot_ind)/mean=1.63%
```

```
Calculating Qdielectric_main for mode 1 (1/1)
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The series.append method is deprecated and will be removed from pandas in a future
```

```
version. Use pandas.concat instead.
```

```
sol = sol.append(self.get_Qdielectric(
```

```
p_dielectric_main_1 = 0.9217873613714703
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for empty
```

```
Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.
```

```
options=pd.Series(get_instance_vars(self.options)),
```

```
WARNING 12:39AM [__init__]: <p>Error: <class 'IndexError'></p>
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_quantum_analysis.py:715: FutureWarning: Support for multi-
```

```
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be removed in
```

```
a future version. Convert to a numpy array before indexing instead.
```

```
result['Q_coupling'] = self.Qm_coupling[vary][self.
```

```
Qm_coupling[vary]
```

```
.columns[junctions]] [modes]#TODO change the columns to junctions
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_quantum_analysis.py:720: FutureWarning: Support for multi-
```

```
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be removed in
```

```
a future version. Convert to a numpy array before indexing instead.
```

```
result['Qs'] =
self.Qs[vary][self.PM[vary].columns[junctions]][modes] #TODO
→ change
the columns to junctions
```

ANALYSIS DONE. Data saved to:

C:\data-pyEPR\Project1\TwoTransmons_hfss\2022-04-26 00-39-29.npz

Differences in variations:

.....
→ ..

Variation 0

Starting the diagonalization

Finished the diagonalization

Pm_norm=

modes

0 1.032643

1 1.033297

dtype: float64

```
Pm_norm idx =
      jj1    jj2
0   True  False
1  False  True
*** P (participation matrix, not normlzd.)
      jj1      jj2
0  0.964167  0.000366
1  0.000370  0.961774
```

*** S (sign-bit matrix)

s_jj1 s_jj2

0 1 1

1 1 1

*** P (participation matrix, normalized.)

1 0.00037

0.00037 0.99

```
*** Chi matrix 01 PT (MHz)
Diag is anharmonicity, off diag is full cross-Kerr.
 312      0.469
 0.469      311
```

```
*** Chi matrix ND (MHz)
 353      1.05
 1.05      344
```

```
*** Frequencies 01 PT (MHz)
0      5312.617742
1      6451.251796
dtype: float64
```

```
*** Frequencies ND (MHz)
0      5292.632746
1      6435.413224
dtype: float64
```

```
*** Q_coupling
Empty DataFrame
Columns: []
Index: [0, 1]
```

3.14.0.3.2 Mode frequencies (MHz)

Numerical diagonalization

```
Lj1      13
0      5292.63
1      6435.41
```

3.14.0.3.3 Kerr Non-linear coefficient table (MHz)

Numerical diagonalization

```
          0      1
Lj1
13  0  352.72    1.05
     1      1.05  343.75
```

Release Ansys's session

```
[51]: eig_2qb.sim.close()
```

(optional) **final wrap:** Close the gui by removing the # in the line below.

```
[52]: gui.main_window.close()
```



[52] : True

[] :

3.14.0.4 1. load fluxonium cell Q3d simulation results

Loading the Maxwell capacitance matrices for the design as shown in the screenshot below: where we have a transmon coupled to a fluxonium through a direct coupler.

For a simple introduction on Maxwell capacitance matrix, check out the following resources:
https://www.fastfieldsolvers.com/Papers/The_Maxwell_Capacitance_Matrix_WP110301_R02.pdf

```
[2]: path = './Fluxonium_8p5MHz_cmat.txt'
flux_mat, _, _, _ = load_q3d_capacitance_matrix(path)
```

```
Imported capacitance matrix with UNITS: [fF] now converted to USER UNITS:
→[fF]
from file:
./Fluxonium_8p5MHz_cmat.txt

<pandas.io.formats.style.Styler at 0x12d1c28c2e0>
```

3.14.0.5 load transmon cell Q3d simulation results

```
[3]: path = './Transmon_5p5GHz_fQ_cmat.txt'
transmon_mat, _, _, _ = load_q3d_capacitance_matrix(path)
```

```
Imported capacitance matrix with UNITS: [fF] now converted to USER UNITS:
→[fF]
from file:
./Transmon_5p5GHz_fQ_cmat.txt

<pandas.io.formats.style.Styler at 0x12d1c584f40>
```

3.14.0.6 2. Create LOM cells from capacitance matrices

3.14.0.6.1 Setting cell objects corresponding to the capacitance simulation results

`coupler_pad_Q1` and `coupler_pad_Q2` refer to the same node corresponding to the direct coupler between the qubits but are different names in the capacitance matrix results file. In order to merge the two capacitance matrices in the LOM analysis, we need to rename them to be the same name.

The following three parameters, `ind_dict`, `jj_dict`, `cj_dict`, all have the same structure. Each is a dictionary where the keys are tuples, giving the nodes that a junction is in between, and the values specifying the relevant values associated with the junction. `ind_dict` lets you specify the junction inductance in nH; `jj_dict` specifies the Josephson junction name (you can give the junction any name you wish; just need to be consistent with the name); `cj_dict` specifies the junction capacitance in fF. In the case of the fluxonium, we will set E_j and E_l directly later instead of deriving from the junction inductance; and since we are only

concerned with capacitive coupling here (what's currently supported), `ind_dict` can just be a placeholder whose actual value is not important.

```
[4]: # cell 1

opt1 = dict(
    node_rename = {'coupling_pad_Q1': 'coupling'},
    cap_mat = flux_mat,
    ind_dict = {('pad_bot_Q1', 'pad_top_Q1'): 1}, # placeholder
    ↪ inductance here; only used for node-basis transformation and reduction
    jj_dict = {('pad_bot_Q1', 'pad_top_Q1'): 'j1'},
)
cell_1 = Cell(opt1)

# cell 2
opt2 = dict(
    node_rename = {'coupling_pad_Q2': 'coupling'},
    cap_mat = transmon_mat,
    ind_dict = {('pad_bot_Q2', 'pad_top_Q2'): 12.31},
    jj_dict = {('pad_bot_Q2', 'pad_top_Q2'): 'j2'},
)
cell_2 = Cell(opt2)
```

3.14.0.7 3. Create subsystems

3.14.0.7.1 Creating the four subsystems, corresponding to the 2 qubits

Subsystem takes three required arguments. The four currently supported system types are `TRANSMON`, `FLUXONIUM`, `TL_RESONATOR` (transmission line resonator) and `LUMPED_RESONATOR`. `nodes` lets you specify which node the subsystem should be mapped to in the cells. They should be consistent with the node names you have given previously. `q_opts` lets specify any optional parameters you want to give. For example, for the fluxonium, you can provide `scqubits` parameters such as `EJ`, `EL` and `flux` here.

```
[5]: # subsystem 1: fluxonium
fluxonium = Subsystem(name='fluxonium', sys_type='FLUXONIUM',
    ↪ nodes=['j1'], q_opts={'EJ':4860, 'EL':1140, 'flux': .5})

# subsystem 2: transmon
transmon = Subsystem(name='transmon', sys_type='TRANSMON', nodes=['j2'],
    ↪ q_opts={'ncut': 150, 'truncated_dim':10})
```

3.14.0.8 4. Create the composite system from the cells and the subsystems

```
[6]: composite_sys = CompositeSystem(  
    subsystems=[fluxonium, transmon],  
    cells=[cell_1, cell_2],  
    grd_node='ground_main_plane')
```

The `circuitGraph` object encapsulates the lumped model circuit analysis (i.e., LOM analysis) and contain the intermediate as well as final L and C matrices, their inverses needed to construct the Hamiltonian of the composite system. For more details on the meaning and calculation of these matrices, check out <https://arxiv.org/pdf/2103.10344.pdf>.

Just to note that you can use the analysis without needing to know any detail about this object.

```
[7]: cg = composite_sys.circuitGraph()  
print(cg)
```

```
node_jj_basis:  
-----  
['j1', 'pad_top_Q1', 'j2', 'pad_top_Q2', 'coupling']  
  
nodes_keep:  
-----  
['j1', 'j2']  
  
L_inv_k (reduced inverse inductance matrix):  
-----  


|    |     |          |
|----|-----|----------|
|    | j1  | j2       |
| j1 | 1.0 | 0.000000 |
| j2 | 0.0 | 0.081235 |

  
C_k (reduced capacitance matrix):  
-----  


|    |           |           |
|----|-----------|-----------|
|    | j1        | j2        |
| j1 | 21.769175 | -0.249771 |
| j2 | -0.249771 | 58.195757 |


```

3.14.0.9 5. Generate the hilberspace from the composite system, leveraging the scqubits package

`add_interaction()` adds the interaction terms between the subsystems. Currently, capacitive coupling is supported (which is extracted by from off-diagonal elements in the C matrices, see *eqn 12, 13* in <https://arxiv.org/pdf/2103.10344.pdf>) and contribute to the interaction.

```
[8]: hilbertspace = composite_sys.create_hilbertspace()
      hilbertspace = composite_sys.add_interaction()
      print(hilbertspace)
```

HilbertSpace: subsystems

```
Fluxonium-----| [Fluxonium_2]
  | EJ: 4860
  | EC: 889.8446120935232
  | EL: 1140
  | flux: 0.5
  | cutoff: 110
  | truncated_dim: 10
  |
  | dim: 110
```

```
Transmon-----| [Transmon_2]
  | EJ: 13278.758148398147
  | EC: 332.8624670156746
  | ng: 0.001
  | ncut: 150
  | truncated_dim: 10
  |
  | dim: 301
```

HilbertSpace: interaction terms

```
InteractionTerm-----| [Interaction_1]
  | g_strength: 30.55305847409223
  | operator_list: [(0, array([[ 0.+0.j
-0.-0.44731161j,  0. . . .
  | add_hc: False
```

3.14.0.10 6. Print the results

Print the calculated Hamiltonian parameters from diagonalized composite system Hamiltonian.

The diagonal elements of the χ matrix are the anharmonicities of the respective subsystems and the off-diagonal the dispersive shifts between them.

```
[9]: hamiltonian_results = composite_sys.hamiltonian_results(hilbertspace, evals_count=30)
```

Finished eigensystem.

system frequencies in GHz:

```
{'fluxonium': 0.3601817422174399, 'transmon': 5.591162362502458}
```

Chi matrices in MHz

```
fluxonium      transmon
fluxonium  3293.889946    0.451843
transmon       0.451843 -391.649924
```

```
[10]: hamiltonian_results['chi_in_MHz'].to_dataframe()
```

```
[10]:      fluxonium      transmon
fluxonium  3293.889946    0.451843
transmon       0.451843 -391.649924
```

The χ 's between the subsystems are based on the coupling strengths, g 's between them (which are computed using the coupling capacitance (currently capacitive coupling is supported) and zero point fluctuations of the subsystem's charge operator at the coupling location)

```
[11]: composite_sys.compute_gs()
```

```
[11]:      fluxonium      transmon
fluxonium   0.000000 30.553058
transmon     30.553058  0.000000
```

```
[12]: fluxonium.h_params
```

```
[12]: {'EJ': 4860,
       'EC': 889.8446120935232,
       'EL': 1140,
       'flux': 0.5,
```

```
'Q_zpf': 3.204353268e-19,
'default_charge_op': Operator(op=array([[ 0.+0.j           , -0.-0.
                                         ↪44731161j,
0.+0.j           , ...,
0.+0.j           , 0.+0.j           , 0.+0.j           ],
[ 0.+0.44731161j, 0.+0.j           , -0.-0.63259415j, ...,
0.+0.j           , 0.+0.j           , 0.+0.j           ],
[ 0.+0.j           , 0.+0.63259415j, 0.+0.j           , ...,
0.+0.j           , 0.+0.j           , 0.+0.j           ],
...,
[ 0.+0.j           , 0.+0.j           , 0.+0.j           , ...,
0.+0.j           , -0.-4.64859865j, 0.+0.j           ],
[ 0.+0.j           , 0.+0.j           , 0.+0.j           , ...,
0.+4.64859865j, 0.+0.j           , -0.-4.67007036j],
[ 0.+0.j           , 0.+0.j           , 0.+0.j           , ...,
0.+0.j           , 0.+4.67007036j, 0.+0.j           ],
↪add_hc=False)}
```

[13]: transmon.h_params

```
{'EJ': 13278.758148398147,
'EC': 332.8624670156746,
'Q_zpf': 3.204353268e-19,
'default_charge_op': Operator(op=array([-150,      0,      0, ...,      0,      0,
                                         ↪0,
0],
[ 0, -149,      0, ...,      0,      0,      0],
[ 0,      0, -148, ...,      0,      0,      0],
...,
[ 0,      0,      0, ..., 148,      0,      0],
[ 0,      0,      0, ...,      0, 149,      0],
[ 0,      0,      0, ...,      0,      0, 150]), add_hc=False)}
```

3.14.0.11 7. let's sweep some parameters now

Let's sweep the flux from 0 to 1 in a grid of 100 points in unit of flux quantum using scQubits's sweeping library.

```
[14]: _sys = hilbertspace.subsys_list[0] # fluxonium

def update_hilbertspace(param_val):
    _sys.flux = param_val

param_name = 'flux'
```

```
param_vals = np.linspace(0, 1, 101)

sweep = scq.ParameterSweep(
    paramvals_by_name={param_name: param_vals},
    evals_count=30,
    hilbertspace=hilbertspace,
    subsys_update_info={param_name: [_sys]},
    update_hilbertspace=update_hilbertspace,
)
```

```
Bare spectra: 0% | 0/101 [00:00<?, ?it/s]
Bare spectra: 0% | 0/1 [00:00<?, ?it/s]
Dressed spectrum: 0% | 0/101 [00:00<?, ?it/s]
```

3.14.0.11.1 Plot transition frequencies as a function of the flux

0->1 transition for the transmon

0->1 transition for the fluxonium

0->2 transition for the fluxonium

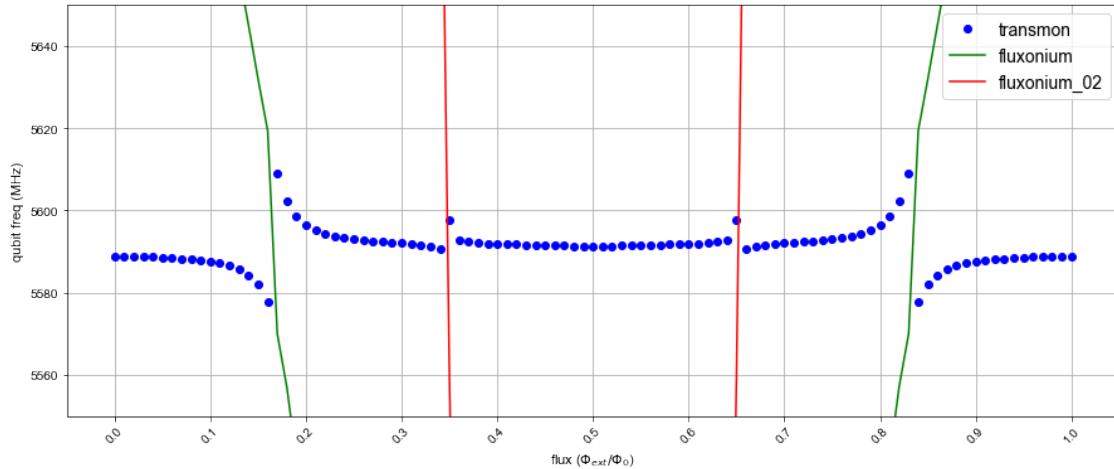
```
[15]: wq_t = sweep.transitions(False, [], (0, 0), (0, 1))[1][0]
wq_f = sweep.transitions(False, [], (0, 0), (1, 0))[1][0]
wq_f_02 = sweep.transitions(False, [], (0, 0), (2, 0))[1][0]

plt.figure(figsize=(15, 6))
plt.plot(param_vals, wq_t, 'ob', label='transmon')
plt.plot(param_vals, wq_f, 'g-', label='fluxonium')
plt.plot(param_vals, wq_f_02, 'r-', label='fluxonium_02')

plt.xticks(param_vals[::10], rotation=45)
plt.xlabel(r'flux ($\Phi_{\text{ext}}/\Phi_0$)')
plt.ylabel(r'qubit freq (MHz)')
plt.ylim([5550, 5650])

plt.grid()
plt.legend(fontsize=14)
```

[15]: <matplotlib.legend.Legend at 0x12d1e325c70>



3.14.0.11.2 The dispersive shift, χ between the two qubits as a function of the flux

```
[16]: wq_f = sweep.transitions(False, [], (0, 0), (1, 0))[1][0]
wq_f_t = sweep.transitions(False, [], (0, 1), (1, 1))[1][0]

chi = wq_f_t - wq_f

plt.figure(figsize=(15, 6))
plt.plot(param_vals, chi, 'ob')

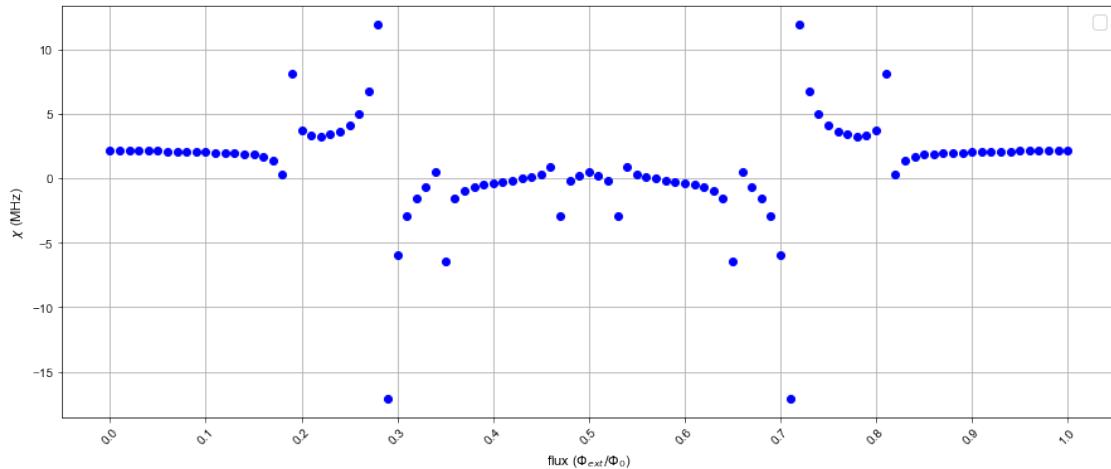
plt.xticks(param_vals[::10], rotation=45)

plt.xlabel(r'flux ($\Phi_{ext}/\Phi_0$)')
plt.ylabel(r'$\chi$ (MHz)')
# plt.ylim([5550, 5650])

plt.grid()
plt.legend(fontsize=14)
```

WARNING:matplotlib.legend:No artists with labels found to put in legend. ↴
 ↴Note
 that artists whose label start with an underscore are ignored when legend() ↴
 ↴is
 called with no argument.

```
[16]: <matplotlib.legend.Legend at 0x12d1e62c910>
```



3.14.0.11.3 Zooming on the fluxonium sweet spot: its 0->1 transition as a function of the flux

```
[17]: wq_f = sweep.transitions(False, [], (0, 0), (1, 0))[1][0]
```

```
plt.figure(figsize=(15, 6))
plt.plot(param_vals, wq_f, 'ob')

plt.xticks(param_vals[::10], rotation=45)

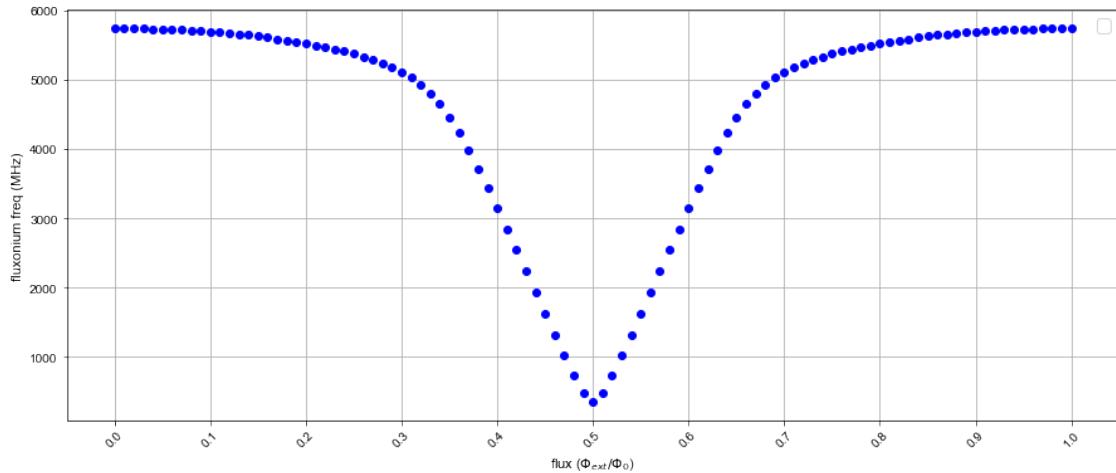
plt.xlabel(r'flux ($\Phi_{ext}/\Phi_0$)')
plt.ylabel(r'fluxonium freq (MHz)')

plt.grid()
plt.legend(fontsize=14)
```

WARNING:matplotlib.legend:No artists with labels found to put in legend. ↴
 ↴Note

that artists whose label start with an underscore are ignored when legend() ↴
 ↴is
 called with no argument.

```
[17]: <matplotlib.legend.Legend at 0x12d1e7ee460>
```



```
[18]: scq.get_units()
scq.set_units('MHz')
```

WARNING:py.warnings:UserWarning: Changing units (by calling set_units())
 ↪after
 initializing qubit instances is likely to cause unintended inconsistencies.
 C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\scqubits\core\units.
 ↪py:
 47

```
[18]: 'MHz'
```

3.14.0.11.4 Coherences of the fluxonium as a function of the flux

For more information on their calculations and assumptions made, check out `scqubits` documentation: <https://scqubits.readthedocs.io/en/latest/guide/guide-noise.html>

```
[19]: _sys.plot_t1_effective_vs_paramvals(param_name='flux',
                                         param_vals=param_vals)
```

Spectral data: 0% | 0/101 [00:00<?, ?it/s]

WARNING:py.warnings:UserWarning: By default all methods that involve calculations of the t1 coherence times/rates, return a sum of upward (i.e., excitation), and downward (i.e., relaxation) rates. To change this behavior, parameter total=False can be passed to any t1-related coherence methods.
 ↪With

total=False, only a one-directional transition between levels i and j is
 ↪used to
 calculate the required t1 time or rate.

See documentation for details.

This warning can be disabled by executing:
`scqubits.settings.T1_DEFAULT_WARNING=False`

```
C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\scqubits\core\noise.  

→py:  

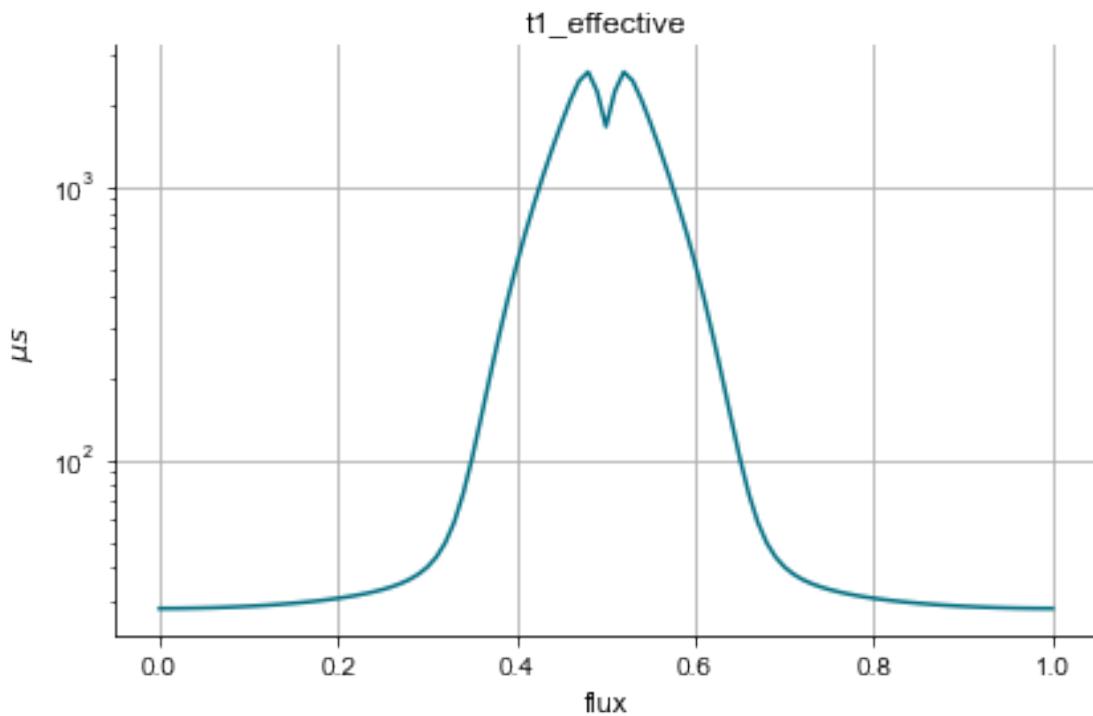
1177
```

```
[19]: (<Figure size 432x288 with 1 Axes>,  

       <AxesSubplot:title={'center':'t1_effective'}, xlabel='flux',  

       →ylabel='$\mu  

      s$'>)
```



```
[20]: sys.plot_coherence_vs_paramvals(param_name='flux', param_vals=param_vals)
```

Spectral data: 0% | 0/101 [00:00<?, ?it/s]

```
[20]: (<Figure size 576x864 with 8 Axes>,  

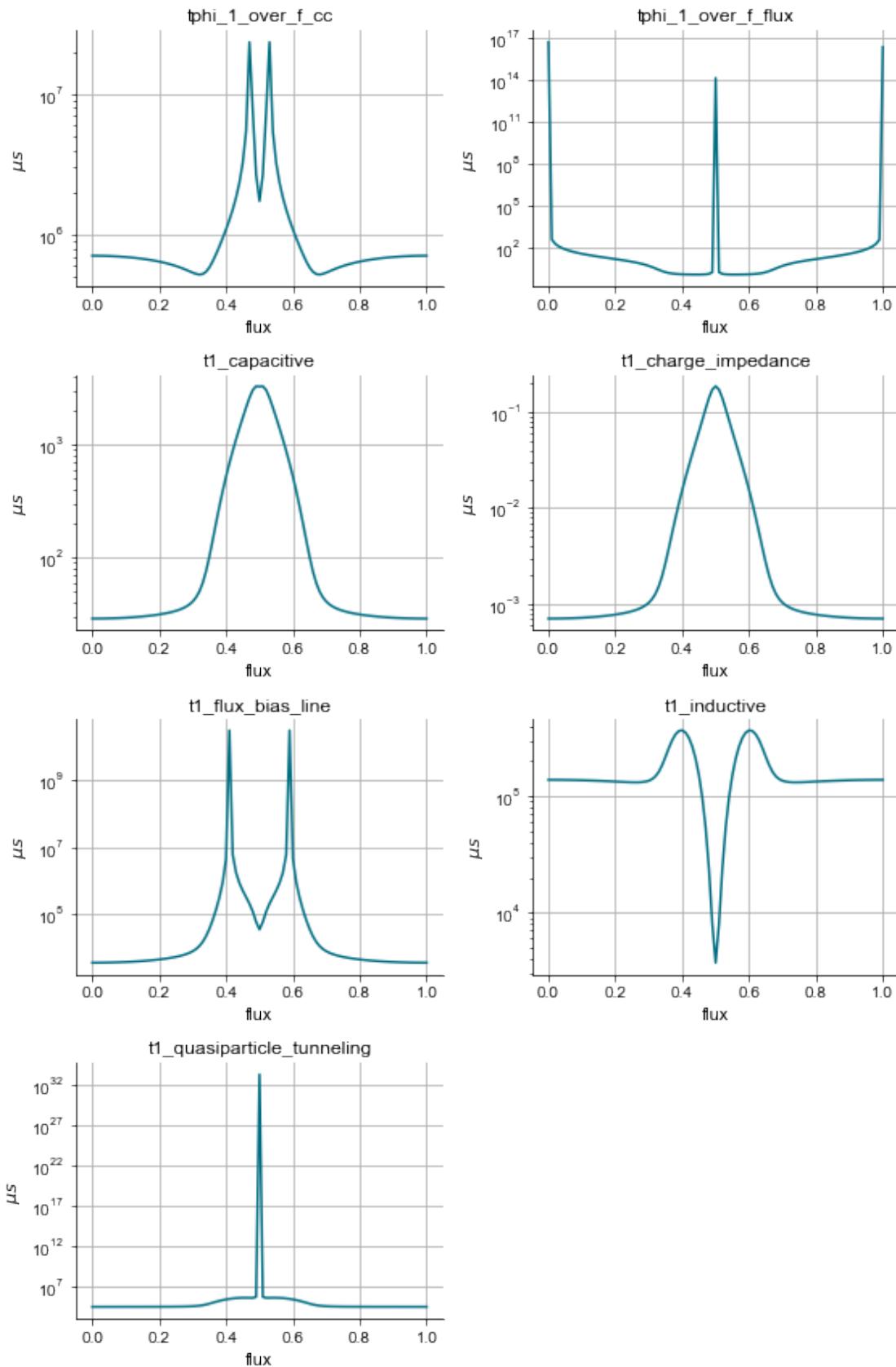
       array([[<AxesSubplot:title={'center':'tphi_1_over_f_cc'}, xlabel='flux',  

              ylabel='$\mu s$',  

              <AxesSubplot:title={'center':'tphi_1_over_f_flux'}, xlabel='flux',  

              →xlabel='flux',
```

```
ylabel='$\mu s$'],
[<AxesSubplot:title={'center':'t1_capacitive'}, xlabel='flux',
ylabel='$\mu s$',
<AxesSubplot:title={'center':'t1_charge_impedance'}, xlabel='flux',
ylabel='$\mu s$'],
[<AxesSubplot:title={'center':'t1_flux_bias_line'}, xlabel='flux',
ylabel='$\mu s$',
<AxesSubplot:title={'center':'t1_inductive'}, xlabel='flux',
ylabel='$\mu s$'],
[<AxesSubplot:title={'center':'t1_quasiparticle_tunneling'},
xlabel='flux', ylabel='$\mu s$',
<AxesSubplot:>]], dtype=object))
```



[]:

3.15 Analyzing and tuning a transmon qubit

We will showcase two methods (EPR and LOM) to analyze the same design. Specifically, we will use here the **advanced** methods to run the simulations and analysis, which directly control renderers and external packages. Please refer to the tutorial notebooks 4.1 and 4.2 to follow the **suggested** flow to run the analysis.

3.15.1 Index

3.15.1.0.1 Transmon design

1. Prepare the single transmon qubit layout in qiskit-metal.

3.15.1.0.2 Transmon analysis using EPR method

1. Set-up and run a finite element simulate to extract the eigenmode.
2. Display EM fields to inspect quality of the setup.
3. Identify junction parameters for the EPR analysis.
4. Run EPR analysis on single eigenmode.
5. Get qubit freq and anharmonicity.
6. Calculate EPR of substrate.

3.15.1.0.3 Transmon analysis using LOM method

1. Calculate the capacitance matrix.
2. Execute analysis on extracted LOM.

3.15.2 Prerequisite

You need to have a working local installation of Ansys. Also you will need the following directives and imports.

```
[1]: %reload_ext autoreload
%autoreload 2

import qiskit_metal as metal
from qiskit_metal import designs, draw
from qiskit_metal import MetalGUI, Dict, Headings
import pyEPR as epr
```

3.16 1. Create the Qbit design

Fix the design dimensions that you intend to reflect in the design rendering. Note that the design size extends from the origin into the first quadrant.

```
[2]: design = designs.DesignPlanar({}, True)
design.chips.main.size['size_x'] = '2mm'
design.chips.main.size['size_y'] = '2mm'

gui = MetalGUI(design)
```

Create a single transmon in the center of the chip previously defined.

```
[3]: from qiskit_metal qlibrary.qubits.transmon_pocket import TransmonPocket

design.delete_all_components()

q1 = TransmonPocket(design, 'Q1', options = dict(
    pad_width = '425 um',
    pocket_height = '650um',
    connection_pads=dict(
        readout = dict(loc_W=+1, loc_H=+1, pad_width='200um')
    )))
    )
    )

gui.rebuild()
gui.autoscale()
```

3.17 2. Analyze the transmon using the Eigenmode-EPR method

In this section we will use a semi-manual (advanced) analysis flow. Please refer to tutorial 4.2 for the suggested method. As illustrated, the methods are equivalent, but the advanced method allows you to directly override some renderer-specific settings.

3.17.0.1 Finite Element Eigenmode Analysis

3.17.0.1.1 Setup

Select the analysis you intend to run from the `qiskit_metal.analyses` collection. Select the design to analyze and the tool to use for any external simulation.

```
[4]: from qiskit_metal.analyses.quantization import EPRanalysis
eig_qb = EPRanalysis(design, "hfss")
```

For the Eigenmode simulation portion, you can either: 1. Use the `eig_qb` user-friendly methods (see tutorial 4.2) 2. Control directly the simulation tool from the tool's GUI (outside metal - see specific vendor instructions) 3. Use the renderer methods In this section we show the advanced method (method 3).

The renderer can be reached from the analysis class. Let's give it a shorter alias.

```
[5]: hfss = eig_qb.sim.renderer
```

Now we connect to the tool using the unified command.

```
[6]: hfss.start()
```

```
INFO 12:56AM [connect_project]: Connecting to Ansys Desktop API...
INFO 12:56AM [load_ansys_project]:      Opened Ansys App
INFO 12:56AM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0
INFO 12:56AM [load_ansys_project]:      Opened Ansys Project
    Folder:  C:/Users/Bartu/Documents/Ansoft/
    Project: Project2
INFO 12:56AM [connect_design]: No active design found (or error getting ↴
    ↵active
design).
INFO 12:56AM [connect]:           Connected to project "Project2". No design
detected
```

```
[6]: True
```

The previous command is supposed to open ansys (if closed), create a new project and finally connect this notebook to it.

If for any reason the previous cell failed, please try the manual path described in the next three cells: 1. uncomment and execute only **one** of the lines in the first cell. 1. uncomment and execute the second cell. 1. uncomment and execute only **one** of the lines in the third cell.

```
[ ]: # hfss.open_ansys()  # this opens Ansys 2021 R2 if present
# hfss.open_ansys(path_var='ANSYSEM_ROOT211')
# hfss.open_ansys(path='C:\\Program Files\\AnsysEM\\AnsysEM21.1\\Win64')
# hfss.open_ansys(path='../../..\\Program Files\\AnsysEM\\AnsysEM21.1\\Win64'
```

```
[ ]: # hfss.new_ansys_project()
```

```
[ ]: # hfss.connect_ansys()
# hfss.connect_ansys('C:\\project_path\\', 'Project1') # will open a ↴
    ↵saved project before linking the Jupyter session
```

3.17.0.1.2 Execute simulation and verify convergence

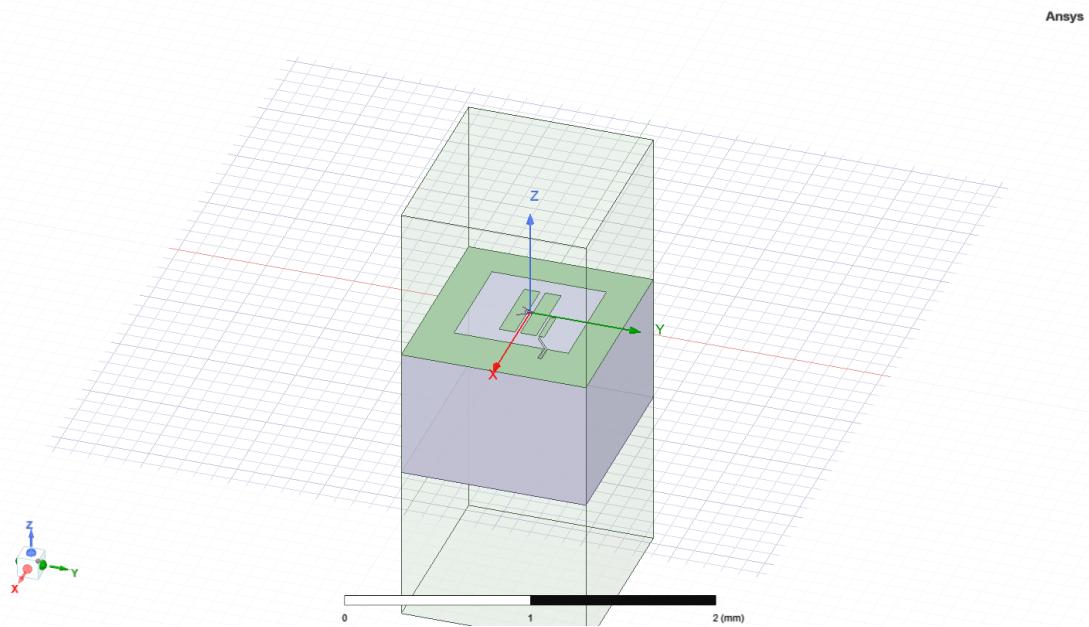
Create and activate an eigenmode design called “TransmonQubit”.

```
[7]: hfss.activate_ansys_design("TransmonQubit", 'eigenmode') # use ↴
    ↵new_ansys_design() to force creation of a blank design
```

```
12:57AM 10s WARNING [activate_ansys_design]: The design_name=TransmonQubit was
not in active project. Designs in active project are:
[] . A new design will be added to the project.
INFO 12:57AM [connect_design]: Opened active design
    Design: TransmonQubit [Solution type: Eigenmode]
WARNING 12:57AM [connect_setup]: No design setup detected.
WARNING 12:57AM [connect_setup]: Creating eigenmode default setup.
INFO 12:57AM [get_setup]: Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
```

Render the single qubit in Metal, called Q1, to “TransmonQubit” design in Ansys.

```
[8]: hfss.render_design(['Q1'], [])
hfss.save_screenshot()
```



```
[8]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/B.
Advanced - Direct use of the renderers/ansys.png')
```

Set the convergence parameters and junction properties in the Ansys design. Then run the analysis and plot the convergence.

```
[9]: # Analysis properties
setup = hfss.pinfo.setup
setup.passes = 10
print(f""")
```

```

Number of eigenmodes to find      = {setup.n_modes}
Number of simulation passes      = {setup.passes}
Convergence freq max delta percent diff = {setup.delta_f}
""")

pinfo = hfss.pinfo
pinfo.design.set_variable('Lj', '10 nH')
pinfo.design.set_variable('Cj', '0 fF')

setup.analyze()

```

INFO 12:57AM [analyze]: Analyzing setup Setup

```

Number of eigenmodes to find      = 1
Number of simulation passes      = 10
Convergence freq max delta percent diff = 0.1

```

To plot the results you can use the `plot_convergences()` method from the `eig_qb.sim` object. The method will read the data from the variables local to the `eig_qb.sim` object, so we first need to assign the simulation results to these two variables. let's do both (assignment and plotting) in the next cell.

```
[10]: eig_qb.sim.convergence_t, eig_qb.sim.convergence_f, _ = hfss.
       ↪get_convergences()
eig_qb.sim.plot_convergences()
```

```

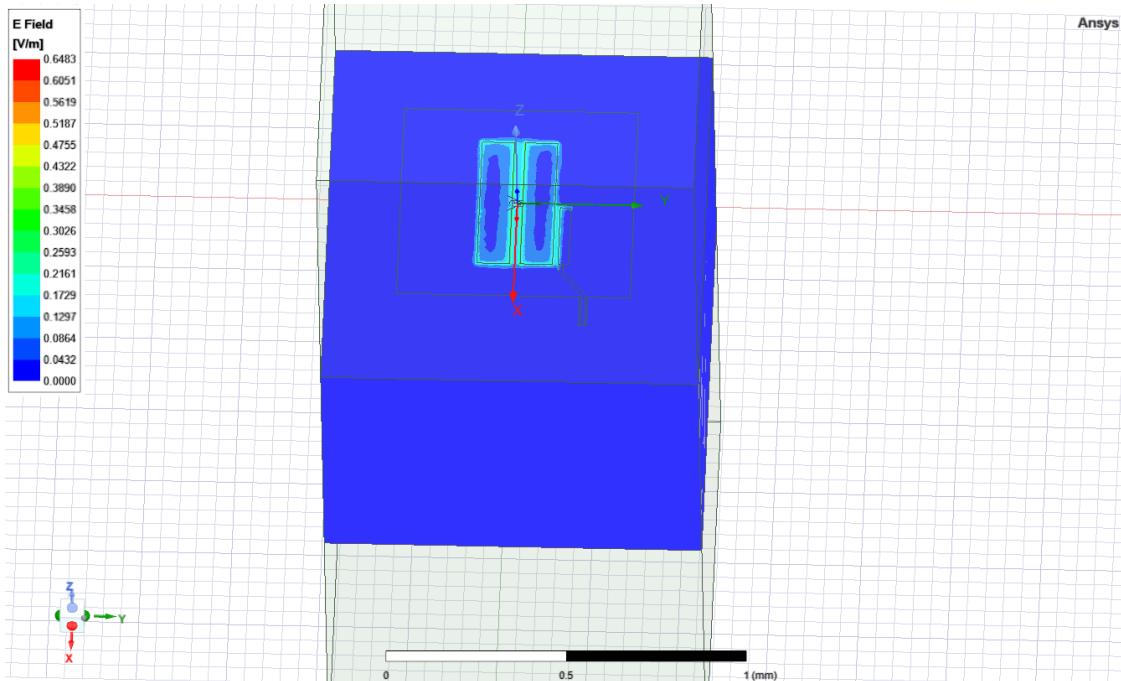
12:59AM 09s INFO [get_f_convergence]: Saved convergences to
C:\Users\Bartu\Desktop\qiskit-metal\tutorials\4 Analysis\B. Advanced -_
↪Direct
use of the renderers\hfss_eig_f_convergence.csv

```

3.17.0.1.3 Plot the EM field for inspection

Display the Ansys modeler window and plot the E-field on the chip's surface.

```
[11]: hfss.modeler._modeler.ShowWindow()
hfss.plot_fields('main')
hfss.save_screenshot()
```



[11]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/B.
Advanced - Direct use of the renderers/ansys.png')

Delete the newly created E-field plot to prepare for the next phase.

[13]: `hfss.clear_fields(['Mag_E1'])`

3.17.0.2 EPR Analysis

In the suggested (tutorial 4.2) flow, we would now prepare the setup using `eig_qb.setup` and run the analysis with `eig_qb.run_epr()`. Notice that this method requires previous set of the `eig_qb` variables `convergence_t` and `convergence_f` like we did a few cells earlier.

However we here exemplify the advanced approach, which is Ansys-specific since it uses the `pyEPR` module methods directly. ##### Setup Identify the non-linear (Josephson) junctions in the model. You will need to list the junctions in the epr setup.

In this case there's only one junction, namely 'jj'. Let's see what we need to change in the default setup.

[14]: `pinfo = hfss.pinfo
pinfo.junctions['jj'] = {'Lj_variable': 'Lj', 'rect': 'JJ_rect_Lj_Q1_rect_jj',
'line': 'JJ_Lj_Q1_rect_jj', 'Cj_variable': 'Cj'}`

```
pinfo.validate_junction_info() # Check that valid names of variables and objects have been supplied
pinfo.dissipative['dielectrics_bulk'] = ['main'] # Dissipative elements: specify
```

3.17.0.2.1 Execute the energy distribution analysis

Execute microwave analysis on eigenmode solutions.

```
[15]: eprd = epr.DistributedAnalysis(pinfo)
```

```
Design "TransmonQubit" info:
    # eigenmodes      1
    # variations      1
```

Find the electric and magnetic energy stored in the substrate and the system as a whole.

```
[16]: _elec = eprd.calc_energy_electric()
_elec_substrate = eprd.calc_energy_electric(None, 'main')
_mag = eprd.calc_energy_magnetic()

print(f"""
_elec_all      = {_elec}
_elec_substrate = {_elec_substrate}
EPR of substrate = {_elec_substrate / _elec * 100 :.1f}%

_mag       = {_mag}
""")
```

```
_elec_all      = 1.789039739382e-24
_elec_substrate = 1.64736636730922e-24
EPR of substrate = 92.1%

_mag       = 4.63056036295054e-26
```

3.17.0.2.2 Run the EPR analysis

Perform EPR analysis for all modes and variations.

```
[17]: eprd.do_EPR_analysis()
```

```
# 4a. Perform Hamiltonian spectrum post-analysis, building on mw solutions using EPR
epra = epr.QuantumAnalysis(eprd.data_filename)
```

```

epra.analyze_all_variations(cos_trunc = 8, fock_trunc = 7)

# 4b. Report solved results
swp_variable = 'Lj' # suppose we swept an optimetric analysis vs. □
→ inductance Lj_alice
epra.plot_hamiltonian_results(swp_variable=swp_variable)
epra.report_results(swp_variable=swp_variable, numeric=True)

```

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.
 options=pd.Series(get_instance_vars(self.options)),

Variation 0 [1/1]

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1096: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

Ljs = pd.Series({})

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1097: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

Cjs = pd.Series({})

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

_Om = pd.Series({})

Mode 0 at 6.36 GHz [1/1]

Calculating _magnetic,

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

```

Sj = pd.Series({})

_electric
    (_E-_H)/_E      _E      _H
    97.4%  8.945e-25 2.315e-26

Calculating junction energy participation ration (EPR)
method='line_voltage'. First estimates:
junction      EPR p_0j   sign s_0j   (p_capacitive)
Energy fraction (Lj over Lj&Cj)= 96.90%
jj            0.99457 (+)      0.031769
(U_tot_cap-U_tot_ind)/mean=0.55%

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.

Qp = pd.Series({})

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The
series.append method is deprecated and will be removed from pandas in ared
future
version. Use pandas.concat instead.

sol = sol.append(self.get_Qdielectric()

Calculating Qdielectric_main for mode 0 (0/0)
p_dielectric_main_0 = 0.9208103828248559

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\project_info.py:226: FutureWarning: The default dtype forred
empty
Series will be 'object' instead of 'float64' in a future version. Specify a
dtype explicitly to silence this warning.

options=pd.Series(get_instance_vars(self.options)),

WARNING 01:00AM [__init__]: <p>Error: <class 'IndexError'></p>
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_quantum_analysis.py:715: FutureWarning: Support forred
multi-
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will bered
removed in
a future version. Convert to a numpy array before indexing instead.

```

```
result['Q_coupling'] = self.Qm_coupling[varyation][self.
    ↪Qm_coupling[varyation]
.columns[junctions]][modes]#TODO change the columns to junctions

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_quantum_analysis.py:720: FutureWarning: Support for
    ↪multi-
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be
    ↪removed in
a future version. Convert to a numpy array before indexing instead.
result['Qs'] =
self.Qs[varyation][self.PM[varyation].columns[junctions]][modes] #TODO
    ↪change
the columns to junctions
```

ANALYSIS DONE. Data saved to:

C:\data-pyEPR\Project2\TransmonQubit\2022-04-28 01-00-32.npz

Differences in variations:

.....
↪ ..

Variation 0

Starting the diagonalization
Finished the diagonalization

Pm_norm=
modes
0 1.011234
dtype: float64

Pm_norm idx =
 jj
0 True
*** P (participation matrix, not normlz.)
 jj
0 0.963947

*** S (sign-bit matrix)

```
s_jj
0      1
*** P (participation matrix, normalized.)
      0.97

*** Chi matrix 01 PT (MHz)
Diag is anharmonicity, off diag is full cross-Kerr.
      294

*** Chi matrix ND (MHz)
      325

*** Frequencies 01 PT (MHz)
0      6066.516345
dtype: float64

*** Frequencies ND (MHz)
0      6051.724691
dtype: float64

*** Q_coupling
Empty DataFrame
Columns: []
Index: [0]
```

3.17.0.2.3 Mode frequencies (MHz)

Numerical diagonalization

```
Lj      10
0      6051.72
```

3.17.0.2.4 Kerr Non-linear coefficient table (MHz)

Numerical diagonalization

```
0
Lj
10 0  325.05
```

Release Ansys session

```
[18]: eig_qb.sim.close()
```

3.18 3. Analyze the transmon using the LOM method

In this section we will use a semi-manual (advanced) analysis flow. Please refer to tutorial 4.1 for the suggested method. As illustrated, the methods are equivalent, but the advanced method allows you to directly override some renderer-specific settings.

3.18.0.1 Capacitance matrix extraction

3.18.0.1.1 Setup

Select the analysis you intend to run from the `qiskit_metal.analyses` collection. Select the design to analyze and the tool to use for any external simulation.

```
[19]: from qiskit_metal.analyses.quantization import LOManalysis  
c1 = LOManalysis(design, "q3d")
```

For the capacitive simulation portion, you can either: 1. Use the `c1` user-friendly methods (see tutorial 4.1) 2. Control directly the simulation tool from the tool's GUI (outside metal - see specific vendor instructions) 3. Use the renderer methods In this section we show the advanced method (method 3).

The renderer can be reached from the analysis class. Let's give it a shorter alias.

```
[20]: q3d = c1.sim.renderer
```

Now we connect to the simulation tool, similarly to what we have done for the eigenmode analysis.

```
[21]: q3d.start()
```

```
INFO 01:01AM [connect_project]: Connecting to Ansys Desktop API...  
INFO 01:01AM [load_ansys_project]:      Opened Ansys App  
INFO 01:01AM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0  
INFO 01:01AM [load_ansys_project]:      Opened Ansys Project  
    Folder:  C:/Users/Bartu/Documents/Ansoft/  
    Project: Project2  
INFO 01:01AM [connect_design]:  Opened active design  
    Design:  TransmonQubit [Solution type: Eigenmode]  
INFO 01:01AM [get_setup]:      Opened setup `Setup` (<class  
'pyEPR.ansys.HfssEMSetup'>)  
INFO 01:01AM [connect]:      Connected to project "Project2" and design  
"TransmonQubit"
```

```
[21]: True
```

If the simulator is already open, the line above will simply connect to the open session, project and design.

3.18.0.1.2 Execute simulation and verify convergence

Create and activate a q3d design called “TransmonQubit_q3d”.

```
[22]: q3d.activate_ansys_design("TransmonQubit_q3d", 'capacitive') # use
      →new_ansys_design() to force creation of a blank design
```

```
01:01AM 54s WARNING [activate_ansys_design]: The
      ↵design_name=TransmonQubit_q3d
was not in active project. Designs in active project are:
['TransmonQubit']. A new design will be added to the project.
INFO 01:01AM [connect_design]: Opened active design
      Design: TransmonQubit_q3d [Solution type: Q3D]
WARNING 01:01AM [connect_setup]: No design setup detected.
WARNING 01:01AM [connect_setup]: Creating Q3D default setup.
INFO 01:01AM [get_setup]: Opened setup `Setup` (<class
'pyEPR.ansys.AnsysQ3DSetup'>)
```

Next, we render the qubit to Ansys Q3D for analysis. We set the readout pin of the qubit in the ‘open’ termination list of the render so its capacitance is properly simulated.

```
[23]: q3d.render_design(['Q1'], [('Q1', 'readout')])
```

Execute the capacitance extraction and verify convergence. This cell analyzes the default setup.

```
[24]: q3d.analyze_setup("Setup")
```

```
INFO 01:02AM [get_setup]: Opened setup `Setup` (<class
'pyEPR.ansys.AnsysQ3DSetup'>)
INFO 01:02AM [analyze]: Analyzing setup Setup
```

This simulation had 4 nets, the two charge islands of the floating transmon, the readout coupler, and the ground, resulting in a 4x4 capacitance matrix. Output is of type DataFrame.

```
[25]: c1.sim.capacitance_matrix, c1.sim.units = q3d.get_capacitance_matrix()
c1.sim.capacitance_all_passes, _ = q3d.get_capacitance_all_passes()
c1.sim.capacitance_matrix
```

```
INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpzi7ld05w.txt, C, , Setup:LastAdaptive,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 1, False
INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp4f1i7i8d.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 1, False
```

```

INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpjw3t4qrg.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 2, False
INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpfobr6k4v.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 3, False
INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpixsd55g6.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 4, False
INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpgt55gwgt.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 5, False
INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpmeozuers.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 6, False
INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpp8om5prc.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 7, False
INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpsjjr6v5o.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 8, False
INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpi_p18wdg.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 9, False
INFO 01:04AM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp5no8ky10.txt, C, , Setup:AdaptivePass,
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 10, False

```

[25] :

	ground_main_plane	pad_bot_Q1	pad_top_Q1	\
ground_main_plane	177.78538	-44.74611	-38.34722	
pad_bot_Q1	-44.74611	82.84141	-32.48515	
pad_top_Q1	-38.34722	-32.48515	93.38943	
readout_connector_pad_Q1	-37.03041	-2.30760	-19.67329	
		readout_connector_pad_Q1		
ground_main_plane		-37.03041		
pad_bot_Q1		-2.30760		
pad_top_Q1		-19.67329		
readout_connector_pad_Q1		60.14998		

3.18.0.2 LOM Analysis

Now we provide the junction lumped element values, and complete the analysis by plotting the convergence. This is the same steps used in the suggested flow from tutorial 4.1.

```
[26]: c1.setup.junctions=Dict(Lj=12.31, Cj=2)
c1.setup.freq_readout = 7.0
c1.setup.freq_bus = []

c1.run_lom()
c1.lumped_oscillator_all
```

[1, 2] [3]

Predicted Values

Transmon Properties

f_Q 5.492369 [GHz]
 EC 320.340887 [MHz]
 EJ 13.273404 [GHz]
 alpha -374.552102 [MHz]
 dispersion 58.767574 [KHz]
 Lq 12.305036 [nH]
 Cq 60.467548 [fF]
 T1 47.110126 [us]

Coupling Properties

tCqbus1 -7.535751 [fF]
 gbus1_in_MHz -119.111599 [MHz]
 chi_bus1 -3.815566 [MHz]
 1/T1bus1 3378.359561 [Hz]
 T1bus1 47.110126 [us]

Bus-Bus Couplings

```
[26]:      fQ          EC          EJ          alpha      dispersion \
1  5.848077  366.598874  13.273404 -435.018704  183.524692
2  5.800483  360.198512  13.273404 -426.553995  158.893606
3  5.728978  350.706154  13.273404 -414.05886  127.373567
4  5.645726  339.839946  13.273404 -399.840701  97.757395
5  5.574615  330.715418  13.273404 -387.971176  77.483483
6  5.552051  327.850119  13.273404 -384.256913  71.883362
7  5.522295  324.093652  13.273404 -379.396816  65.050231
8  5.508895  322.410099  13.273404 -377.222089  62.166575
9  5.492369  320.340887  13.273404 -374.552102  58.767574
```

	gbus	chi_in_MHz	chi_r MHz	gr MHz
1	[-120.90670176407029]	[-7.037262053036213]	7.037262	120.906702
2	[-112.5714316750082]	[-5.610863434478079]	5.610863	112.571432
3	[-114.4721344200509]	[-5.13583772345763]	5.135838	114.472134

4	[-113.75236108111946]	[-4.422671613573631]	4.422672	113.752361
5	[-115.42485160814473]	[-4.067027529551065]	4.067028	115.424852
6	[-116.90656488361248]	[-4.027920633489713]	4.027921	116.906565
7	[-118.53068129631139]	[-3.9548493935624265]	3.954849	118.530681
8	[-118.93049750280109]	[-3.9007913880709433]	3.900791	118.930498
9	[-119.11159900747404]	[-3.8155660691693987]	3.815566	119.111599

And plot the convergence.

[27]:
c1.plot_convergence();
c1.plot_convergence_chi()

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:503: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.  
    self._hfss_variables[varyation] = pd.Series(
```

```
INFO 01:04AM [hfss_report_full_convergence]: Creating report for variation 0  
Design "TransmonQubit_q3d" info:  
    # eigenmodes      0  
    # variations      1
```

Release the simulator and close the analysis.

[28]: c1.sim.close()

(optional) close the GUI.

[29]: gui.main_window.close()

[29]: True

[]:

3.19 Analyzing and tuning a resonator

We will use here the advanced EPR method to run the simulation and analysis, which directly controls renderers and external packages. Please refer to the tutorial notebook 4.2 to follow the suggested flow to run the analysis.

3.19.0.1 Resonator design

1. Prepare the single transmon qubit layout in qiskit-metal.

3.19.0.2 Resonator analysis using EPR method

1. Set-up and run a finite element simulate to extract the eigenmode.
2. Display EM fields to inspect quality of the setup.
3. Calculate EPR of substrate.

3.19.1 Prerequisite

You need to have a working local installation of Ansys. Also you will need the following directives and imports.

```
[1]: %load_ext autoreload
%autoreload 2

import qiskit_metal as metal
from qiskit_metal import designs, draw
from qiskit_metal import MetalGUI, Dict, Headings
import pyEPR as epr
```

3.20 1. Create the Resonator design

Fix the design dimensions that you intend to reflect in the design rendering. Note that the design size extends from the origin into the first quadrant.

```
[2]: design = designs.DesignPlanar({}, True)
design.chips.main.size['size_x'] = '2mm'
design.chips.main.size['size_y'] = '2mm'

gui = MetalGUI(design)
```

Create a readout resonator. Here, we define one end of the resonator as a short and the other end as an open.

```
[3]: from qiskit_metal qlibrary.tlines.meandered import RouteMeander
from qiskit_metal qlibrary terminations.open_to_ground import OpenToGround
```

```

from qiskit_metal qlibrary terminations short_to_ground import
    ShortToGround

design.delete_all_components()

otg = OpenToGround(design, 'open_to_ground', options=dict(pos_x='1.25mm',
    pos_y='0um', orientation='0'))
stg = ShortToGround(design, 'short_to_ground', options=dict(pos_x='-1.
    -25mm', pos_y='0um', orientation='180'))
rt_meander = RouteMeander(design, 'readout', Dict(
    total_length='6 mm',
    hfss_wire_bonds = True,
    fillet='90 um',
    lead = dict(start_straight='100um'),
    pin_inputs=Dict(
        start_pin=Dict(component='short_to_ground', pin='short'),
        end_pin=Dict(component='open_to_ground', pin='open'))))

gui.rebuild()
gui.autoscale()

```

3.21 2. Analyze the resonator using the Eigenmode-EPR method

In this section we will use a semi-manual (advanced) analysis flow. Please refer to tutorial 4.2 for the suggested method. As illustrated, the methods are equivalent, but the advanced method allows you to directly override some renderer-specific settings.

3.21.0.1 Finite Element Eigenmode Analysis

3.21.0.1.1 Setup

Select the analysis you intend to run from the `qiskit_metal.analyses` collection. Select the design to analyze and the tool to use for any external simulation.

[4]:

```

from qiskit_metal.analyses.quantization import EPRanalysis
eig_res = EPRanalysis(design, "hfss")

```

For the Eigenmode simulation portion, you can either: 1. Use the `eig_res` user-friendly methods (see tutorial 4.2) 2. Control directly the simulation tool from the tool's GUI (outside metal - see specific vendor instructions) 3. Use the renderer methods In this section we show the advanced method (method 3).

The renderer can be reached from the analysis class. Let's give it a shorter alias.

```
[5]: hfss = eig_res.sim.renderer
```

Now we connect to the tool using the unified command.

```
[6]: hfss.start()
```

```
INFO 09:51PM [connect_project]: Connecting to Ansys Desktop API...
INFO 09:51PM [load_ansys_project]:      Opened Ansys App
INFO 09:51PM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0
INFO 09:51PM [load_ansys_project]:      Opened Ansys Project
    Folder:   C:/Users/Bartu/Documents/Ansoft/
    Project:  Project8
INFO 09:51PM [connect_design]: No active design found (or error getting ↴active
design).
INFO 09:51PM [connect]:           Connected to project "Project8". No design
detected
```

```
[6]: True
```

The previous command is supposed to open ansys (if closed), create a new project and finally connect this notebook to it.

If for any reason the previous cell failed, please try the manual path described in the next three cells: 1. uncomment and execute only **one** of the lines in the first cell. 1. uncomment and execute the second cell. 1. uncomment and execute only **one** of the lines in the third cell.

```
[7]: # hfss.open_ansys()  # this opens Ansys 2021 R2 if present
# hfss.open_ansys(path_var='ANSYSEM_ROOT211')
# hfss.open_ansys(path='C:\Program Files\AnsysEM\AnsysEM21.1\Win64')
# hfss.open_ansys(path='../../../../../Program Files/AnsysEM/AnsysEM21.1/Win64')
```

```
[8]: # hfss.new_ansys_project()
```

```
[9]: # hfss.connect_ansys()
# hfss.connect_ansys('C:\\project_path\\', 'Project1') # will open a ↴saved project before linking the Jupyter session
```

3.21.0.1.2 Execute simulation and verify convergence

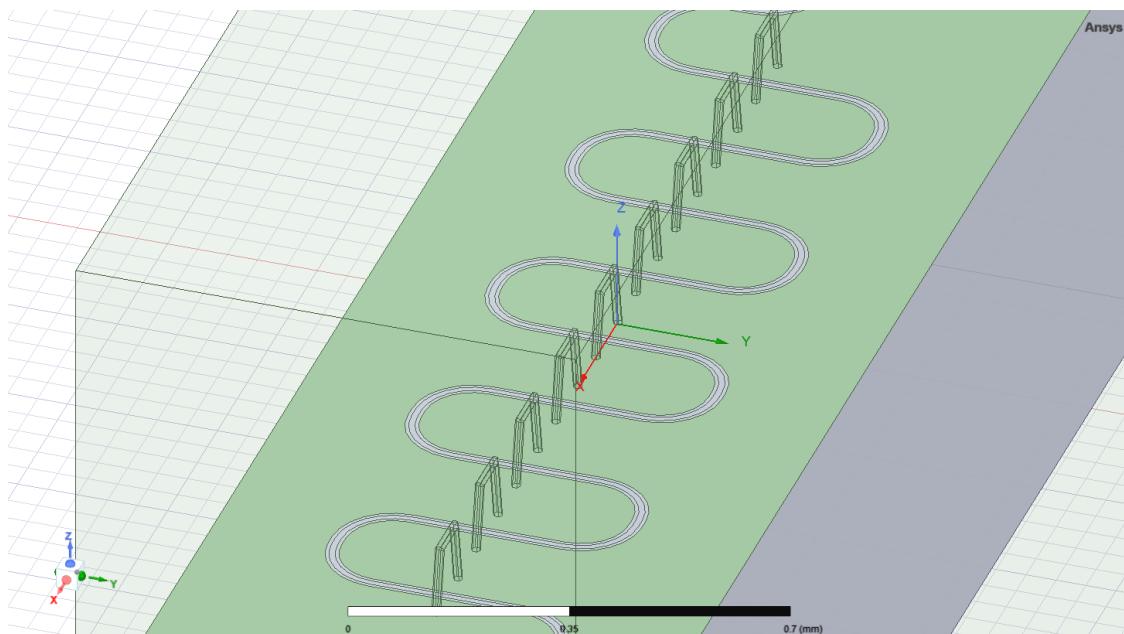
Create and activate an eigenmode design called “Readout”.

```
[10]: hfss.activate_ansys_design("Readout", 'eigenmode') # use ↴new_ansys_design() to force creation of a blank design
```

```
09:54PM 54s WARNING [activate_ansys_design]: The design_name=Readout was not in
active project. Designs in active project are:
[] . A new design will be added to the project.
INFO 09:55PM [connect_design]: Opened active design
    Design: Readout [Solution type: Eigenmode]
WARNING 09:55PM [connect_setup]: No design setup detected.
WARNING 09:55PM [connect_setup]: Creating eigenmode default setup.
INFO 09:55PM [get_setup]: Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
```

Render the resonator called readout in Metal, to “Readout” design in Ansys.

```
[11]: hfss.render_design(['short_to_ground', 'readout', 'open_to_ground'], [])
hfss.save_screenshot()
```



```
[11]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/B.
Advanced - Direct use of the renderers/ansys.png')
```

Set the convergence parameters and junction properties in the Ansys design. Then run the analysis and plot the convergence.

```
[12]: # Analysis properties
setup = hfss.pinfo.setup
setup.passes = 20
print(f"""
```

```

Number of eigenmodes to find      = {setup.n_modes}
Number of simulation passes      = {setup.passes}
Convergence freq max delta percent diff = {setup.delta_f}
""")

# Next 2 lines are counterintuitive, since there is no junction in this
# resonator.
# However, these are necessary to make pyEPR work correctly. Please do
# note delete
hfss.pinfo.design.set_variable('Lj', '10 nH')
hfss.pinfo.design.set_variable('Cj', '0 fF')

setup.analyze()

```

INFO 09:56PM [analyze]: Analyzing setup Setup

```

Number of eigenmodes to find      = 1
Number of simulation passes      = 20
Convergence freq max delta percent diff = 0.1

```

To plot the results you can use the `plot_convergences()` method from the `eig_res.sim` object. The method will read the data from the variables local to the `eig_res.sim` object, so we first need to assign the simulation results to these two variables. let's do both (assignment and plotting) in the next cell.

Note: if `hfss.get_convergences()` raises a `com_error`, it is likely due to the simulation not converging. Try increasing the number of passes in the setup above or tweak other simulation or layout parameters.

[13] : `eig_res.sim.convergence_t, eig_res.sim.convergence_f, _ = hfss.`
 `→get_convergences()`
`eig_res.sim.plot_convergences()`

```

09:59PM 29s INFO [get_f_convergence]: Saved convergences to
C:\Users\Bartu\Desktop\qiskit-metal\tutorials\4 Analysis\B. Advanced -_
→Direct
use of the renderers\hfss_eig_f_convergence.csv

```

3.21.0.1.3 Refine the resonator design, rerun simulation and verify convergence

Next, we change the length of the resonator and see how the eigen frequency changes.

[14] : `rt_meander.options.total_length = '9 mm'`
`gui.rebuild()`

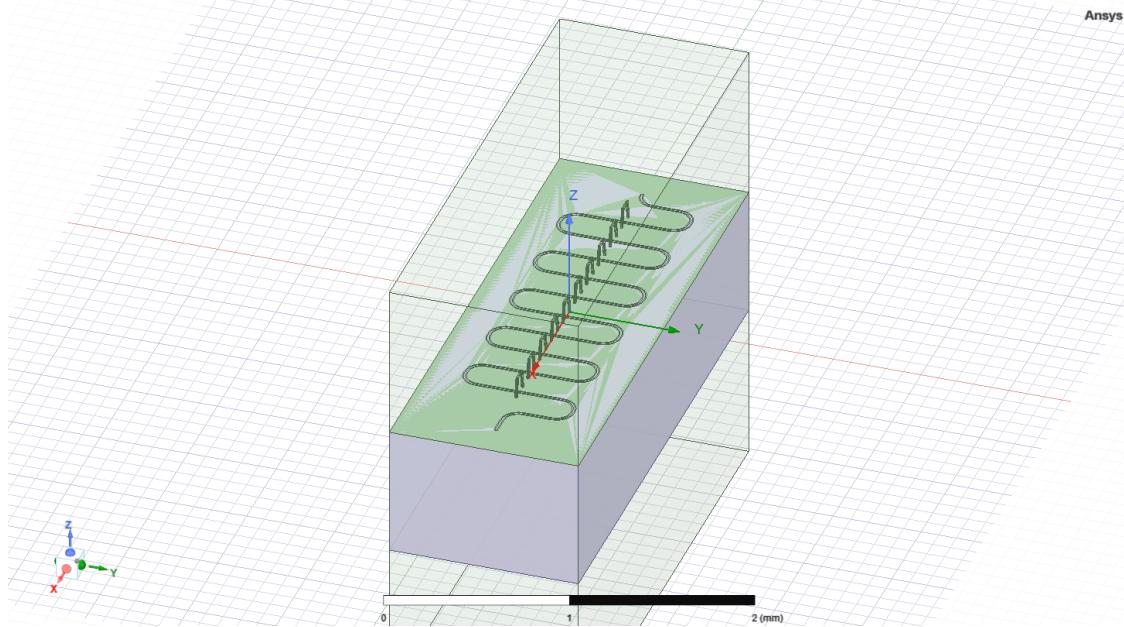
```
gui.autoscale()
```

Need to render the updated design again. Let's do that in a new design ("Readout_1") to avoid conflicts. Alternatively you will need to delete all the shapes from the previous design to be able to re-draw in it.

```
[15]: hfss.activate_ansys_design("Readout_1", 'eigenmode') # use
      ↪new_ansys_design() to force creation of a blank design
```

```
10:00PM 01s WARNING [activate_ansys_design]: The design_name=Readout_1 was
↪not
in active project. Designs in active project are:
['Readout']. A new design will be added to the project.
INFO 10:00PM [connect_design]: Opened active design
    Design: Readout_1 [Solution type: Eigenmode]
WARNING 10:00PM [connect_setup]: No design setup detected.
WARNING 10:00PM [connect_setup]: Creating eigenmode default setup.
INFO 10:00PM [get_setup]: Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
```

```
[16]: hfss.render_design(['short_to_ground', 'readout', 'open_to_ground'], [])
hfss.save_screenshot()
```



```
[16]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/B.
Advanced - Direct use of the renderers/ansys.png')
```

now re-execute the analysis. We will skip the design variable setup since they are still in memory.

```
[17]: # Analysis properties
setup = hfss.pinfo.setup
setup.passes = 20
print(f"""
Number of eigenmodes to find      = {setup.n_modes}
Number of simulation passes      = {setup.passes}
Convergence freq max delta percent diff  = {setup.delta_f}
""")

# Next 2 lines are counterintuitive, since there is no junction in this
# resonator.
# However, these are necessary to make pyEPR work correctly. Please do
# note delete
hfss.pinfo.design.set_variable('Lj', '10 nH')
hfss.pinfo.design.set_variable('Cj', '0 fF')

setup.analyze()
```

INFO 10:00PM [analyze]: Analyzing setup Setup

```
Number of eigenmodes to find      = 1
Number of simulation passes      = 20
Convergence freq max delta percent diff  = 0.1
```

```
[18]: eig_res.sim.convergence_t, eig_res.sim.convergence_f, _ = hfss.
       get_convergences()
eig_res.sim.plot_convergences()
```

10:03PM 42s INFO [get_f_convergence]: Saved convergences to
 C:\Users\Bartu\Desktop\qiskit-metal\tutorials\4 Analysis\B. Advanced -
 Direct
 use of the renderers\hfss_eig_f_convergence.csv

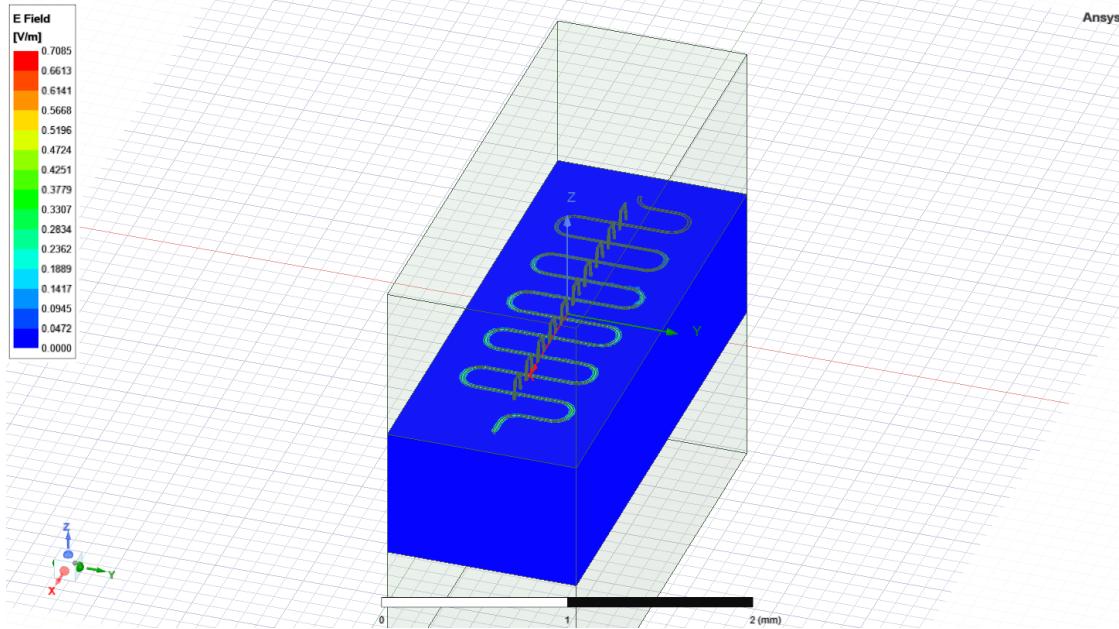
From the above analyses we observe that for a total length of 6 mm for the resonator, the Eigen Frequency was close to 4.9 GHz. However, for a total length of 9 mm, this frequency is close to 3.3 GHz. Similar analysis can be performed by the user for matching a particular frequency of interest.

3.21.0.1.4 Plot the EM field for inspection

Display the Ansys modeler window and plot the E-field on the chip's surface.

```
[19]: hfss.modeler._modeler.ShowWindow()
hfss.plot_ansys_fields('main')
hfss.save_screenshot()
```

10:03PM 44s WARNING [plot_ansys_fields]: This method is deprecated. Change [your](#) scripts to use plot_fields()



```
[19]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/B.
Advanced - Direct use of the renderers/ansys.png')
```

Delete the newly created E-field plot to prepare for the next phase.

```
[20]: hfss.plot_ansys_delete(['Mag_E1'])
```

10:03PM 46s WARNING [plot_ansys_delete]: This method is deprecated. Change [your](#) scripts to use clear_fields()

3.21.0.2 EPR Analysis

In the **suggested** (tutorial 4.2) flow, we would now prepare the setup using `eig_res.setup` and run the analysis with `eig_res.run_epr()`. Notice that this method requires previous set of the `eig_res` variables `convergence_t` and `convergence_f` like we did a few cells earlier.

However we here exemplify the advanced approach, which is Ansys-specific since it uses the pyEPR module methods directly. ##### Execute the energy distribution analysis

First initialize epr for non-junction analysis. This will set the ground plain to 'main'.

```
[21]: eig_res.epr_start(no_junctions=True)
```

```
Design "Readout_1" info:  
    # eigenmodes      1  
    # variations      1
```

```
[21]: {'junctions': {}, 'dissipatives': {'dielectrics_bulk': ['main']}}}
```

Execute microwave analysis on eigenmode solutions.

```
[22]: eprd = hfss.epr_distributed_analysis
```

Find the electric and magnetic energy stored in the substrate and the system.

```
[23]: _elec = eprd.calc_energy_electric()  
_elec_substrate = eprd.calc_energy_electric(None, 'main')  
_mag = eprd.calc_energy_magnetic()  
  
print(f"""  
_elec_all      = {_elec}  
_elec_substrate = {_elec_substrate}  
EPR of substrate = {_elec_substrate / _elec * 100 :.1f}%  
  
_mag      = {_mag}  
""")
```

```
_elec_all      = 2.9152283797599e-24  
_elec_substrate = 2.673136978658e-24  
EPR of substrate = 91.7%  
  
_mag      = 2.9152291544723e-24
```

Release Ansys session

```
[24]: eig_res.sim.close()
```

(optional) close the GUI.

```
[25]: gui.main_window.close()
```



[25] : True

[] :

3.22 Analyzing and tuning a transmon qubit with a resonator

We will showcase two methods (EPR and LOM) to analyze the same design. Specifically, we will use here the `advanced` methods to run the simulations and analysis, which directly control renderers and external packages.

3.22.1 Index

3.22.1.0.1 Transmon & resonator design

1. Prepare the single transmon qubit layout in qiskit-metal.

3.22.1.0.2 Transmon & resonator analysis using EPR method

1. Set-up and run a finite element simulate to extract the eigenmode.
2. Display EM fields to inspect quality of the setup.
3. Identify junction parameters for the EPR analysis.
4. Run EPR analysis on single eigenmode.
5. Get qubit freq and anharmonicity.
6. Calculate EPR of substrate.

3.22.1.0.3 Transmon & resonator analysis using LOM method

1. Calculate the capacitance matrix.
2. Execute analysis on extracted LOM.

3.22.2 Prerequisite

You need to have a working local installation of Ansys. Also you will need the following directives and imports.

```
[1]: %load_ext autoreload
%autoreload 2

import qiskit_metal as metal
from qiskit_metal import designs, draw
from qiskit_metal import MetalGUI, Dict, Headings
import pyEPR as epr
```

3.23 1. Create the Qbit design

Fix the design dimensions that you intend to reflect in the design rendering. Note that the design size extends from the origin into the first quadrant.

```
[2]: design = designs.DesignPlanar({}, True)
design.chips.main.size['size_x'] = '2mm'
design.chips.main.size['size_y'] = '2mm'

gui = MetalGUI(design)
```

Create a single transmon with one readout resonator. Please refer to the notebook tutorials 4.11 ad 4.12 if you're not familiar with the code in the cell below.

```
[3]: from qiskit_metal qlibrary.qubits.transmon_pocket import TransmonPocket
from qiskit_metal qlibrary terminations.open_to_ground import OpenToGround
from qiskit_metal qlibrary.tlines.meandered import RouteMeander

design.delete_all_components()

q1 = TransmonPocket(design, 'Q1', options = dict(
    pad_width = '425 um',
    pocket_height = '650um',
    connection_pads=dict(
        readout = dict(loc_W=+1, loc_H=+1, pad_width='200um')
    )))
otg = OpenToGround(design, 'open_to_ground', options=dict(pos_x='1.75mm', ↴ pos_y='0um', orientation='0'))
readout = RouteMeander(design, 'readout', Dict(
    total_length='6 mm',
    hfss_wire_bonds = True,
    fillet='90 um',
    lead = dict(start_straight='100um'),
    pin_inputs=Dict(
        start_pin=Dict(component='Q1', pin='readout'),
        end_pin=Dict(component='open_to_ground', pin='open')), ))
gui.rebuild()
gui.autoscale()
```

3.24 2. Analyze the transmon & resonator using the Eigenmode-EPR method

In this section we will use a semi-manual (advanced) analysis flow. As illustrated, the methods are equivalent, but the advanced method allows you to directly override some renderer-specific settings.

3.24.0.1 Finite Element Eigenmode Analysis

3.24.0.1.1 Setup

Select the analysis you intend to run from the `qiskit_metal.analyses` collection. Select the design to analyze and the tool to use for any external simulation.

```
[4]: from qiskit_metal.analyses.quantization import EPRanalysis  
eig_qres = EPRanalysis(design, "hfss")
```

For the Eigenmode simulation portion, you can either: 1. Use the `eig_qres` user-friendly methods (see tutorial 4.2) 2. Control directly the simulation tool from the tool's GUI (outside metal - see specific vendor instructions) 3. Use the renderer methods In this section we show the advanced method (method 3).

The renderer can be reached from the analysis class. Let's give it a shorter alias.

```
[5]: hfss = eig_qres.sim.renderer
```

Now we connect to the tool using the unified command.

```
[6]: hfss.start()
```

```
INFO 11:52PM [connect_project]: Connecting to Ansys Desktop API...  
INFO 11:52PM [load_ansys_project]:      Opened Ansys App  
INFO 11:52PM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0  
INFO 11:52PM [load_ansys_project]:      Opened Ansys Project  
    Folder:    C:/Users/Bartu/Documents/Ansoft/  
    Project:   Project11  
INFO 11:52PM [connect_design]: No active design found (or error getting  
    ↵active  
design).  
INFO 11:52PM [connect]:           Connected to project "Project11". No design  
detected
```

```
[6]: True
```

The previous command is supposed to open ansys (if closed), create a new project and finally connect this notebook to it.

If for any reason the previous cell failed, please try the manual path described in the next three cells: 1. uncomment and execute only **one** of the lines in the first cell. 1. uncomment and execute the second cell. 1. uncomment and execute only **one** of the lines in the third cell.

```
[7]: # hfss.open_ansys()    # this opens Ansys 2021 R2 if present  
# hfss.open_ansys(path_var='ANSYSEM_ROOT211')  
# hfss.open_ansys(path='C:\Program Files\AnsysEM\AnsysEM21.1\Win64')
```

```
# hfss.open_ansys(path='../../../../Program Files/AnsysEM/AnsysEM21.1/Win64')
```

```
[8]: # hfss.new_ansys_project()
```

```
[9]: # hfss.connect_ansys()  
# hfss.connect_ansys('C:\\\\project_path\\\\', 'Project1') # will open a  
→saved project before linking the Jupyter session
```

3.24.0.1.2 Execute simulation and verify convergence

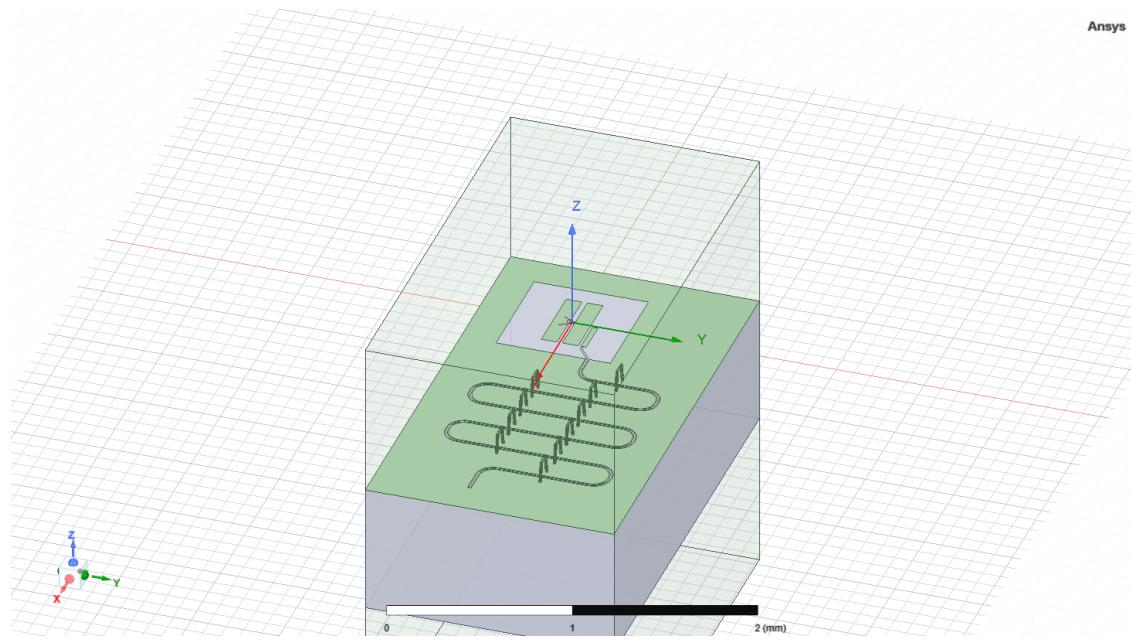
Create and activate an eigenmode design called “TransmonReadout”.

```
[10]: hfss.activate_ansys_design("TransmonReadout", 'eigenmode') # use  
→new_ansys_design() to force creation of a blank design
```

```
11:52PM 56s WARNING [activate_ansys_design]: The  
→design_name=TransmonReadout was  
not in active project. Designs in active project are:  
[]. A new design will be added to the project.  
INFO 11:53PM [connect_design]: Opened active design  
Design: TransmonReadout [Solution type: Eigenmode]  
WARNING 11:53PM [connect_setup]: No design setup detected.  
WARNING 11:53PM [connect_setup]: Creating eigenmode default setup.  
INFO 11:53PM [get_setup]: Opened setup `Setup` (<class  
'pyEPR.ansys.HfssEMSetup'>)
```

Render everything including the qubit and resonator in Metal, to “TransmonReadout” design in Ansys.

```
[11]: hfss.render_design(['Q1', 'readout', 'open_to_ground'], [])  
hfss.save_screenshot()
```



[11]: WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/B.
Advanced - Direct use of the renderers/ansys.png')

Set the convergence parameters and junction properties in the Ansys design. Then run the analysis and plot the convergence.

Note that we seek 2 eigenmodes - one with stronger fields near the transmon, the other with stronger fields near the resonator.

```
[12]: # Analysis properties
setup = hfss.pinfo.setup
setup.n_modes = 2
setup.passes = 10
print(f"""
Number of eigenmodes to find      = {setup.n_modes}
Number of simulation passes       = {setup.passes}
Convergence freq max delta percent diff = {setup.delta_f}
""")

pinfo = hfss.pinfo
pinfo.design.set_variable('Lj', '10 nH')
pinfo.design.set_variable('Cj', '0 fF')

setup.analyze()
```

INFO 11:54PM [analyze]: Analyzing setup Setup

Number of eigenmodes to find = 2
 Number of simulation passes = 10
 Convergence freq max delta percent diff = 0.1

To plot the results you can use the `plot_convergences()` method from the `eig_qres.sim` object. The method will read the data from the variables local to the `eig_qres.sim` object, so we first need to assign the simulation results to these two variables. let's do both (assignment and plotting) in the next cell.

```
[13]: eig_qres.sim.convergence_t, eig_qres.sim.convergence_f, _ = hfss.  

       ↪get_convergences()  

       eig_qres.sim.plot_convergences()
```

```
11:55PM 21s INFO [get_f_convergence]: Saved convergences to  

C:\Users\Bartu\Desktop\qiskit-metal\tutorials\4 Analysis\B. Advanced -  

↪Direct  

use of the renderers\hfss_eig_f_convergence.csv
```

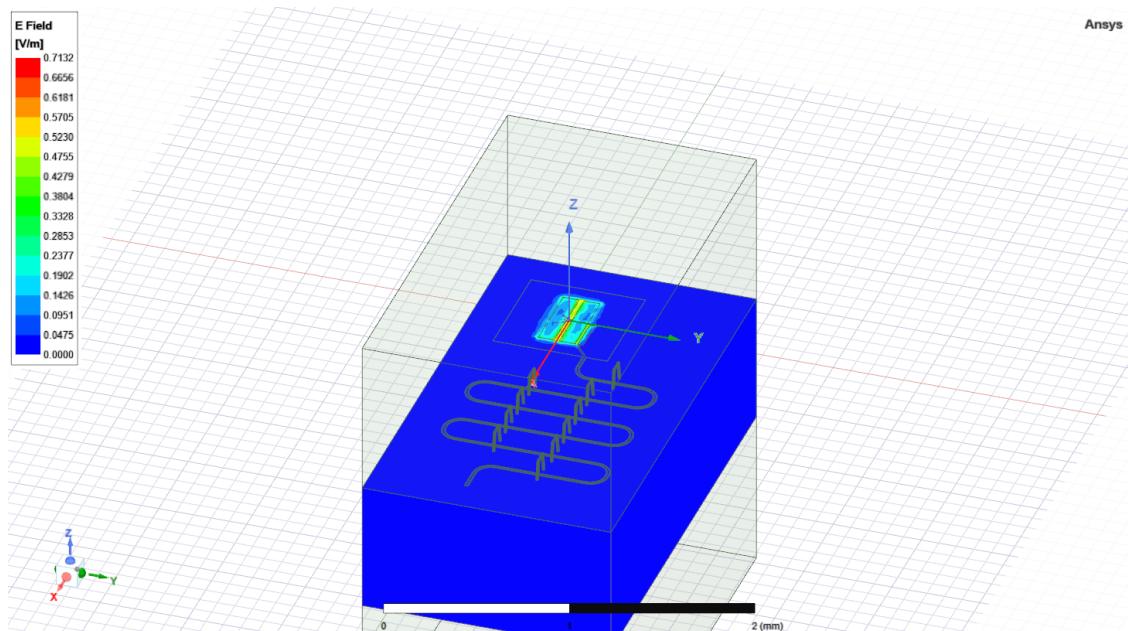
3.24.0.1.3 Plot the EM field for inspection

Display the Ansys modeler window and plot the E-field on the chip's surface.

```
[14]: hfss.modeler._modeler.ShowWindow()  

hfss.plot_fields('main')  

hfss.save_screenshot()
```



[14] : WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/B.
Advanced - Direct use of the renderers/ansys.png')

Delete the newly created E-field plot before moving on.

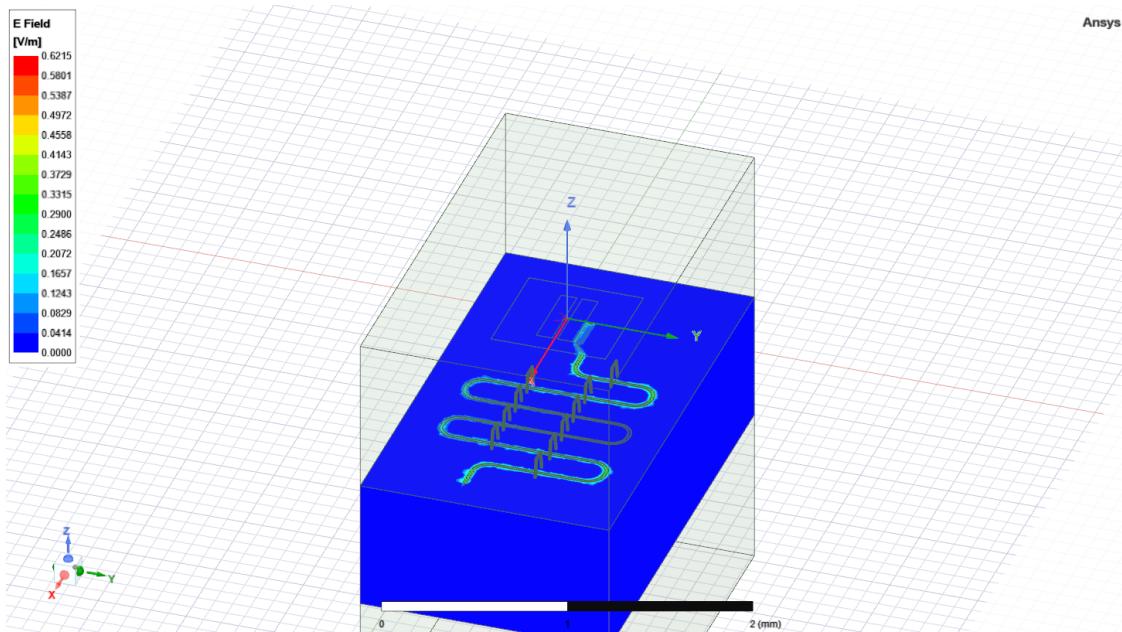
[15] : `hfss.clear_fields(['Mag_E1'])`

To look at the second eigenmode created, we use the following command, and then plot the corresponding E-field.

[16] : `hfss.set_mode(2, "Setup")`

```
INFO 11:55PM [get_setup]:      Opened setup `Setup`  (<class  
'pyEPR.ansys.HfssEMSetup'>)
```

[17] : `hfss.modeler._modeler.ShowWindow()
hfss.plot_fields('main')
hfss.save_screenshot()`



[17] : WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/B.
Advanced - Direct use of the renderers/ansys.png')

We delete this design to prepare for further analysis.

[18] : `hfss.clear_fields(['Mag_E1'])`

3.24.0.2 EPR Analysis

In the suggested (tutorial 4.2) flow, we would now prepare the setup using `eig_qres.setup` and run the analysis with `eig_qres.run_epr()`. Notice that this method requires previous set of the `eig_qres` variables `convergence_t` and `convergence_f` like we did a few cells earlier.

However we here exemplify the advanced approach, which is Ansys-specific since it uses the pyEPR module methods directly. ##### Setup Identify the non-linear (Josephson) junctions in the model. You will need to list the junctions in the epr setup.

In this case there's only one junction, namely 'jj'. Let's see what we need to change in the default setup.

```
[19]: pinfo = hfss.pinfo
pinfo.junctions['jj'] = {'Lj_variable': 'Lj', 'rect': 'JJ_rect_Lj_Q1_rect_jj',
                         'line': 'JJ_Lj_Q1_rect_jj', 'Cj_variable': 'Cj'}
pinfo.validate_junction_info() # Check that valid names of variables and
                               # objects have been supplied
pinfo.dissipative['dielectrics_bulk'] = ['main'] # Dissipative elements:
                                                 # specify
```

3.24.0.2.1 Execute the energy distribution analysis

Execute microwave analysis on eigenmode solutions.

```
[20]: eprd = epr.DistributedAnalysis(pinfo)
```

```
Design "TransmonReadout" info:
    # eigenmodes      2
    # variations      1
```

Find the electric and magnetic energy stored in the substrate and the system as a whole.

```
[21]: _elec = eprd.calc_energy_electric()
_elec_substrate = eprd.calc_energy_electric(None, 'main')
_mag = eprd.calc_energy_magnetic()

print(f"""
_elec_all      = {_elec}
_elec_substrate = {_elec_substrate}
EPR of substrate = {_elec_substrate / _elec * 100 :.1f}%

_mag_all      = {_mag}
_mag % of _elec_all = {_mag / _elec * 100 :.1f}%
```

))))

```
_elec_all      = 3.89576062368317e-24
_elec_substrate = 3.54765069976349e-24
EPR of substrate = 91.1%

_mag_all      = 3.89905831916253e-24
_mag % of _elec_all = 100.1%
```

3.24.0.2.2 Run EPR analysis

Perform EPR analysis for all modes and variations.

[22] : eprd.do_EPR_analysis()

```
# 4a. Perform Hamiltonian spectrum post-analysis, building on mw
→solutions using EPR
epra = epr.QuantumAnalysis(eprd.data_filename)
epra.analyze_all_variations(cos_trunc = 8, fock_trunc = 7)

# 4b. Report solved results
swp_variable = 'Lj' # suppose we swept an optimetric analysis vs.
→inductance Lj_alice
epra.plot_hamiltonian_results(swp_variable=swp_variable)
epra.report_results(swp_variable=swp_variable, numeric=True)
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for
→empty
Series will be 'object' instead of 'float64' in a future version. Specify a
dtype explicitly to silence this warning.
    options=pd.Series(get_instance_vars(self.options)),
```

Variation 0 [1/1]

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1096: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
```

```
Ljs = pd.Series({})
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
```

```
packages\pyEPR\core_distributed_analysis.py:1097: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
```

```
Cjs = pd.Series({})
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
```

```
_Om = pd.Series({})
```

Mode 0 at 6.11 GHz [1/2]

Calculating _magnetic,_electric

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
```

```
Sj = pd.Series({})
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
```

```
Qp = pd.Series({})
```

(_E-_H)/_E	_E	_H
99.2%	7.278e-24	5.719e-26

Calculating junction energy participation ration (EPR)

method='line_voltage'. First estimates:

junction EPR p_0j sign s_0j (p_capacitive)

Energy fraction (Lj over Lj&Cj)= 97.13%

jj 0.991635 (+) 0.0292485

(U_tot_cap-U_tot_ind)/mean=1.47%

Calculating Qdielectric_main for mode 0 (0/1)

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The
series.append method is deprecated and will be removed from pandas in a
future
```

version. Use pandas.concat instead.

```
sol = sol.append(self.get_Qdielectric(
```

```

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
    _Om = pd.Series({})

p_dielectric_main_0 = 0.9117666480938172

Mode 1 at 9.33 GHz [2/2]
    Calculating _magnetic,_electric

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
    Sj = pd.Series({})

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
    Qp = pd.Series({})

        (_E-_H)/_E      _E      _H
        -0.1%  1.948e-24  1.95e-24

Calculating junction energy participation ration (EPR)
    method='line_voltage'. First estimates:
        junction      EPR p_1j   sign s_1j   (p_capacitive)
                    Energy fraction (Lj over Lj&Cj)= 93.57%
        jj            0.00279176 (+)      0.00019198
                    (U_tot_cap-U_tot_ind)/mean=-0.17%
Calculating Qdielectric_main for mode 1 (1/1)

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The
series.append method is deprecated and will be removed from pandas in a
future
version. Use pandas.concat instead.
    sol = sol.append(self.get_Qdielectric(

```

p_dielectric_main_1 = 0.9106439133340368

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for
    ↵empty
Series will be 'object' instead of 'float64' in a future version. Specify a
dtype explicitly to silence this warning.
    options=pd.Series(get_instance_vars(self.options)),

WARNING 11:55PM [__init__]: <p>Error: <class 'IndexError'></p>
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_quantum_analysis.py:715: FutureWarning: Support for
    ↵multi-
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be
    ↵removed in
a future version. Convert to a numpy array before indexing instead.
    result['Q_coupling'] = self.Qm_coupling[varyation][self.
    ↵Qm_coupling[varyation]
.columns[junctions]][modes]#TODO change the columns to junctions

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_quantum_analysis.py:720: FutureWarning: Support for
    ↵multi-
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be
    ↵removed in
a future version. Convert to a numpy array before indexing instead.
    result['Qs'] =
self.Qs[varyation][self.PM[varyation].columns[junctions]][modes] #TODO
    ↵change
the columns to junctions
```

ANALYSIS DONE. Data saved to:

C:\data-pyEPR\Project11\TransmonReadout\2022-05-11 23-55-36.npz

Differences in variations:

Variation 0

```
Starting the diagonalization
Finished the diagonalization
Pm_norm=
modes
0      1.029891
1      0.382121
dtype: float64

Pm_norm idx =
          jj
0    True
1   False
*** P (participation matrix, not normlzd.)
          jj
0  0.963455
1  0.002791

*** S (sign-bit matrix)
s_jj
0      1
1      1
*** P (participation matrix, normalized.)
          0.99
          0.0028

*** Chi matrix 01 PT (MHz)
Diag is anharmonicity, off diag is full cross-Kerr.
          281      2.42
          2.42  0.00519

*** Chi matrix ND (MHz)
          313      2.09
          2.09  0.00389

*** Frequencies 01 PT (MHz)
0      5829.507818
1      9331.184247
dtype: float64

*** Frequencies ND (MHz)
0      5814.793613
1      9331.226443
dtype: float64

*** Q_coupling
```

```
Empty DataFrame
Columns: []
Index: [0, 1]
```

3.24.0.2.3 Mode frequencies (MHz)

Numerical diagonalization

```
Lj      10
0    5814.79
1    9331.23
```

3.24.0.2.4 Kerr Non-linear coefficient table (MHz)

Numerical diagonalization

```
          0           1
Lj
10 0  312.54  2.09e+00
     1      2.09   3.89e-03
```

Release Ansys session

```
[23]: eig_qres.sim.close()
```

3.25 3. Analyze the transmon using the LOM method

In this section we will use a semi-manual (advanced) analysis flow. Please refer to tutorial 4.1 for the **suggested** method. As illustrated, the methods are equivalent, but the advanced method allows you to directly override some renderer-specific settings.

3.25.0.1 Capacitance matrix extraction

3.25.0.1.1 Setup

Select the analysis you intend to run from the `qiskit_metal.analyses` collection. Select the design to analyze and the tool to use for any external simulation.

```
[24]: from qiskit_metal.analyses.quantization import LOManalysis
c2 = LOManalysis(design, "q3d")
```

For the capacitive simulation portion, you can either: 1. Use the `c2` user-friendly methods (see tutorial 4.1) 2. Control directly the simulation tool from the tool's GUI (outside metal - see specific vendor instructions) 3. Use the renderer methods In this section we show the advanced method (method 3).

The renderer can be reached from the analysis class. Let's give it a shorter alias.

[25]: q3d = c2.sim.renderer

Now we connect to the simulation tool, similarly to what we have done for the eigenmode analysis.

[26]: q3d.start()

```
INFO 11:56PM [connect_project]: Connecting to Ansys Desktop API...
INFO 11:56PM [load_ansys_project]:      Opened Ansys App
INFO 11:56PM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0
INFO 11:56PM [load_ansys_project]:      Opened Ansys Project
    Folder:      C:/Users/Bartu/Documents/Ansoft/
    Project:     Project11
INFO 11:56PM [connect_design]:   Opened active design
    Design:     TransmonReadout [Solution type: Eigenmode]
INFO 11:56PM [get_setup]:        Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
INFO 11:56PM [connect]:         Connected to project "Project11" and design
"TransmonReadout"
```

[26]: True

If the simulator is already open, the line above will simply connect to the open session, project and design.

3.25.0.1.2 Execute simulation and verify convergence

Create and activate a q3d design called “TransmonResonator_q3d”.

[27]: q3d.activate_ansys_design("TransmonResonator_q3d", 'capacitive')

```
11:56PM 46s WARNING [activate_ansys_design]: The
design_name=TransmonResonator_q3d was not in active project. Designs in
→active
project are:
['TransmonReadout']. A new design will be added to the project.
INFO 11:56PM [connect_design]:   Opened active design
    Design:     TransmonResonator_q3d [Solution type: Q3D]
WARNING 11:56PM [connect_setup]:      No design setup detected.
WARNING 11:56PM [connect_setup]:      Creating Q3D default setup.
INFO 11:56PM [get_setup]:        Opened setup `Setup` (<class
'pyEPR.ansys.AnsysQ3DSetup'>)
```

Next, we render the existing design to Ansys Q3D for analysis. To ensure that the readout is insulated from the ground plane, we set the ‘readout’ pin of Q1 to have an open termination.

```
[28]: q3d.render_design(['Q1'], [('Q1', 'readout')])
```

Execute the capacitance extraction and verify convergence. This cell analyzes the default setup.

```
[29]: q3d.analyze_setup("Setup")
```

```
INFO 11:56PM [get_setup]:      Opened setup `Setup`  (<class  
'pyEPR.ansys.AnsysQ3DSetup'>)  
INFO 11:56PM [analyze]: Analyzing setup Setup
```

This simulation had 4 nets, the two charge islands of the floating transmon, the readout coupler, and the ground, resulting in a 4x4 capacitance matrix. Output is of type DataFrame.

```
[30]: c2.sim.capacitance_matrix, c2.sim.units = q3d.get_capacitance_matrix()  
c2.sim.capacitance_all_passes, _ = q3d.get_capacitance_all_passes()  
c2.sim.capacitance_matrix
```

```
INFO 11:57PM [get_matrix]: Exporting matrix data to  
(C:\Users\Bartu\AppData\Local\Temp\tmp75uf5zze.txt, C, , Setup:LastAdaptive,  
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 1, False  
INFO 11:57PM [get_matrix]: Exporting matrix data to  
(C:\Users\Bartu\AppData\Local\Temp\_v6rt3c.txt, C, , Setup:AdaptivePass,  
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 1, False  
INFO 11:57PM [get_matrix]: Exporting matrix data to  
(C:\Users\Bartu\AppData\Local\Temp\tmpbjt9v2sy.txt, C, , Setup:AdaptivePass,  
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 2, False  
INFO 11:57PM [get_matrix]: Exporting matrix data to  
(C:\Users\Bartu\AppData\Local\Temp\tmpt86uf193.txt, C, , Setup:AdaptivePass,  
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 3, False  
INFO 11:57PM [get_matrix]: Exporting matrix data to  
(C:\Users\Bartu\AppData\Local\Temp\tmpsaw3uhe3.txt, C, , Setup:AdaptivePass,  
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 4, False  
INFO 11:57PM [get_matrix]: Exporting matrix data to  
(C:\Users\Bartu\AppData\Local\Temp\tmpv3qkts5m.txt, C, , Setup:AdaptivePass,  
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 5, False  
INFO 11:57PM [get_matrix]: Exporting matrix data to  
(C:\Users\Bartu\AppData\Local\Temp\tmpfto5j2np.txt, C, , Setup:AdaptivePass,  
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 6, False  
INFO 11:57PM [get_matrix]: Exporting matrix data to  
(C:\Users\Bartu\AppData\Local\Temp\tmpw8ikc6u_.txt, C, , Setup:AdaptivePass,  
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 7, False  
INFO 11:57PM [get_matrix]: Exporting matrix data to  
(C:\Users\Bartu\AppData\Local\Temp\tmpqy9gsp1o.txt, C, , Setup:AdaptivePass,  
"Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 8, False  
INFO 11:57PM [get_matrix]: Exporting matrix data to
```

(C:\Users\Bartu\AppData\Local\Temp\tmps2gy1_zz.txt, C, , Setup:AdaptivePass,
 "Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 9, False
 INFO 11:57PM [get_matrix]: Exporting matrix data to
 (C:\Users\Bartu\AppData\Local\Temp\tmpk6j3ssvb.txt, C, , Setup:AdaptivePass,
 "Original", "ohm", "nH", "fF", "mSie", 5000000000, Maxwell, 10, False

[30] :

	ground_main_plane	pad_bot_Q1	pad_top_Q1	\
ground_main_plane	177.78538	-44.74611	-38.34722	
pad_bot_Q1	-44.74611	82.84141	-32.48515	
pad_top_Q1	-38.34722	-32.48515	93.38943	
readout_connector_pad_Q1	-37.03041	-2.30760	-19.67329	
				readout_connector_pad_Q1
ground_main_plane		-37.03041		
pad_bot_Q1		-2.30760		
pad_top_Q1		-19.67329		
readout_connector_pad_Q1		60.14998		

3.25.0.2 LOM Analysis

Now we provide the junction lumped element values, and complete the analysis by plotting the convergence. This is the same steps used in the suggested flow from tutorial 4.1.

[31] : c2.setup.junctions=Dict(Lj=12.31, Cj=2)
 c2.setup.freq_readout = 7.0
 c2.setup.freq_bus = []

 c2.run_lom()
 c2.lumped_oscillator_all

[1, 2] [3]
 Predicted Values

Transmon Properties
 f_Q 5.492369 [GHz]
 EC 320.340887 [MHz]
 EJ 13.273404 [GHz]
 alpha -374.552102 [MHz]
 dispersion 58.767574 [KHz]
 Lq 12.305036 [nH]
 Cq 60.467548 [fF]
 T1 47.110126 [us]

Coupling Properties

tCqbus1 -7.535751 [fF]
gbus1_in_MHz -119.111599 [MHz]
chi_bus1 -3.815566 [MHz]
1/T1bus1 3378.359561 [Hz]
T1bus1 47.110126 [us]
Bus-Bus Couplings

[31]:

	fQ	EC	EJ	alpha	dispersion	\
1	5.848077	366.598874	13.273404	-435.018704	183.524692	
2	5.800483	360.198512	13.273404	-426.553995	158.893606	
3	5.728978	350.706154	13.273404	-414.05886	127.373567	
4	5.645726	339.839946	13.273404	-399.840701	97.757395	
5	5.574615	330.715418	13.273404	-387.971176	77.483483	
6	5.552051	327.850119	13.273404	-384.256913	71.883362	
7	5.522295	324.093652	13.273404	-379.396816	65.050231	
8	5.508895	322.410099	13.273404	-377.222089	62.166575	
9	5.492369	320.340887	13.273404	-374.552102	58.767574	

	gbus	chi_in_MHz	xr MHz	gr MHz
1	[-120.90670176407029]	[-7.037262053036213]	7.037262	120.906702
2	[-112.5714316750082]	[-5.610863434478079]	5.610863	112.571432
3	[-114.4721344200509]	[-5.13583772345763]	5.135838	114.472134
4	[-113.75236108111946]	[-4.422671613573631]	4.422672	113.752361
5	[-115.42485160814473]	[-4.067027529551065]	4.067028	115.424852
6	[-116.90656488361248]	[-4.027920633489713]	4.027921	116.906565
7	[-118.53068129631139]	[-3.9548493935624265]	3.954849	118.530681
8	[-118.93049750280109]	[-3.9007913880709433]	3.900791	118.930498
9	[-119.11159900747404]	[-3.8155660691693987]	3.815566	119.111599

And plot the convergence.

[32]:

```
c2.plot_convergence();
c2.plot_convergence_chi()
```

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:503: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

```
self._hfss_variables[variation] = pd.Series(
```

INFO 11:58PM [hfss_report_full_convergence]: Creating report for variation 0

Design "TransmonResonator_q3d" info:

```
# eigenmodes      0
# variations      1
```

Release the simulator and close the analysis.

```
[33]: c2.sim.close()
```

(optional) close the GUI.

```
[ ]: # gui.main_window.close()
```

3.26 Analyzing a double hanger resonator (S Param)

3.26.0.1 Prerequisite

You must have a working local installation of Ansys.

```
[1]: %load_ext autoreload
%autoreload 2

import qiskit_metal as metal
from qiskit_metal import designs, draw
from qiskit_metal import MetalGUI, Dict, Headings
import pyEPR as epr
```

3.26.1 Create the design in Metal

Set up a design of a given dimension. Dimensions will be respected in the design rendering. Note the chip design is centered at origin (0,0).

```
[2]: design = designs.DesignPlanar({}, True)
design.chips.main.size['size_x'] = '2mm'
design.chips.main.size['size_y'] = '2mm'

gui = MetalGUI(design)
```

Perform the necessary imports.

```
[3]: from qiskit_metal qlibrary.couplers.coupled_line_tee import CoupledLineTee
from qiskit_metal qlibrary.tlines.meandered import RouteMeander
from qiskit_metal qlibrary.qubits.transmon_pocket import TransmonPocket
from qiskit_metal qlibrary.tlines.straight_path import RouteStraight
from qiskit_metal qlibrary.terminations.open_to_ground import OpenToGround
```

Add 2 transmons to the design.

```
[4]: options = dict(
    # Some options we want to modify from the defaults
    # (see below for defaults)
    pad_width = '425 um',
    pocket_height = '650um',
    # Adding 4 connectors (see below for defaults)
    connection_pads=dict(
        a = dict(loc_W=+1, loc_H=+1),
        b = dict(loc_W=-1, loc_H=+1, pad_height='30um'),
        c = dict(loc_W=+1, loc_H=-1, pad_width='200um'),
        d = dict(loc_W=-1, loc_H=-1, pad_height='50um')
```

```

        )
    )

## Create 2 transmons

q1 = TransmonPocket(design, 'Q1', options = dict(
    pos_x='+1.4mm', pos_y='0mm', orientation = '90', **options))
q2 = TransmonPocket(design, 'Q2', options = dict(
    pos_x='-0.6mm', pos_y='0mm', orientation = '90', **options))

gui.rebuild()
gui.autoscale()

```

Add 2 hangers consisting of capacitively coupled transmission lines.

```
[5]: TQ1 = CoupledLineTee(design, 'TQ1', options=dict(pos_x='1mm',
                                                    pos_y='3mm',
                                                    coupling_length='200um'))
TQ2 = CoupledLineTee(design, 'TQ2', options=dict(pos_x='1mm',
                                                    pos_y='3mm',
                                                    coupling_length='200um'))

gui.rebuild()
gui.autoscale()
```

Add 2 meandered CPWs connecting the transmons to the hangers.

```
[6]: ops=dict(fillet='90um')
design.overwrite_enabled = True

options1 = Dict(
    total_length='8mm',
    hfss_wire_bonds = True,
    pin_inputs=Dict(
        start_pin=Dict(
            component='TQ1',
            pin='second_end'),
        end_pin=Dict(
            component='Q1',
            pin='a')),
    lead=Dict(
        start_straight='0.1mm'),
    **ops
)
```

```

options2 = Dict(
    total_length='9mm',
    hfss_wire_bonds = True,
    pin_inputs=Dict(
        start_pin=Dict(
            component='TQ2',
            pin='second_end'),
        end_pin=Dict(
            component='Q2',
            pin='a')),
    lead=Dict(
        start_straight='0.1mm'),
    **ops
)

meanderQ1 = RouteMeander(design, 'meanderQ1', options=options1)
meanderQ2 = RouteMeander(design, 'meanderQ2', options=options2)

gui.rebuild()
gui.autoscale()

```

Add 2 open to grounds at the ends of the horizontal CPW.

```
[7]: otg1 = OpenToGround(design, 'otg1', options = dict(pos_x='3mm',
                                                       pos_y='3mm'))
otg2 = OpenToGround(design, 'otg2', options = dict(pos_x = '-3mm',
                                                       pos_y='3mm',
                                                       orientation='180'))

gui.rebuild()
gui.autoscale()
```

Add 3 straight CPWs that comprise the long horizontal CPW.

```
[8]: ops_oR = Dict(hfss_wire_bonds = True,
                  pin_inputs=Dict(
                      start_pin=Dict(
                          component='TQ1',
                          pin='prime_end'),
                      end_pin=Dict(
                          component='otg1',
                          pin='open')))
ops_mid = Dict(hfss_wire_bonds = True,
               pin_inputs=Dict(
                   start_pin=Dict(
```

```

        component='TQ1',
        pin='prime_start'),
    end_pin=Dict(
        component='TQ2',
        pin='prime_end'))
ops_oL = Dict(hfss_wire_bonds = True,
               pin_inputs=Dict(
                   start_pin=Dict(
                       component='TQ2',
                       pin='prime_start'),
                   end_pin=Dict(
                       component='otg2',
                       pin='open')))

cpw_openRight = RouteStraight(design, 'cpw_openRight', options=ops_oR)
cpw_middle = RouteStraight(design, 'cpw_middle', options=ops_mid)
cpw_openLeft = RouteStraight(design, 'cpw_openLeft', options=ops_oL)

gui.rebuild()
gui.autoscale()

```

3.26.2 2. Eigenmode and Impedance analysis using the advanced flow

3.26.2.0.1 Setup

Select the analysis you intend to run from the `qiskit_metal.analyses.simulation` collection. Select the design to analyze and the tool to use for any external simulation.

```
[9]: from qiskit_metal.analyses.simulation.scattering_impedance import_
      ScatteringImpedanceSim
em1 = ScatteringImpedanceSim(design, "hfss")
```

For the DrivenModal simulation portion, you can either: 1. Use the `em1` user-friendly methods 2. Control directly the simulation tool from the tool's GUI (outside metal - see specific vendor instructions) 3. Use the renderer methods In this section we show the advanced method (method 3).

The renderer can be reached from the analysis class. Let's give it a shorter alias.

```
[11]: hfss = em1.renderer
```

Now we connect to the tool using the unified command.

```
[12]: hfss.start()
```

```

INFO 05:00PM [connect_project]: Connecting to Ansys Desktop API...
INFO 05:00PM [load_ansys_project]:      Opened Ansys App
INFO 05:00PM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0
INFO 05:00PM [load_ansys_project]:      Opened Ansys Project
    Folder:   C:/Users/Bartu/Documents/Ansoft/
    Project:  Project11
INFO 05:00PM [connect_design]: No active design found (or error getting
    ↵active
design).
INFO 05:00PM [connect]:           Connected to project "Project11". No design
detected

```

[12]: True

3.26.2.0.2 Execute simulation and verify convergence

Create and activate an eigenmode design called “HangingResonators”.

[13]: `hfss.activate_ansys_design("HangingResonators", 'drivenmodal')`

```

05:01PM 23s WARNING [activate_ansys_design]: The
    ↵design_name=HangingResonators
was not in active project. Designs in active project are:
[] . A new design will be added to the project.
INFO 05:01PM [connect_design]: Opened active design
    Design: HangingResonators [Solution type: DrivenModal]
WARNING 05:01PM [connect_setup]:      No design setup detected.
WARNING 05:01PM [connect_setup]:      Creating drivenmodal default setup.
INFO 05:01PM [get_setup]:      Opened setup `Setup` (<class
    'pyEPR.ansys.HfssDMSetup'>)

```

Set the buffer width at the edge of the design to be 0.5 mm in both directions.

[14]: `hfss.options['x_buffer_width_mm'] = 0.5
hfss.options['y_buffer_width_mm'] = 0.5`

3.26.2.0.3 Execute simulation and observe the Impedance

Assign lumped ports on the two cpw terminations. Then observe the impedance plots.

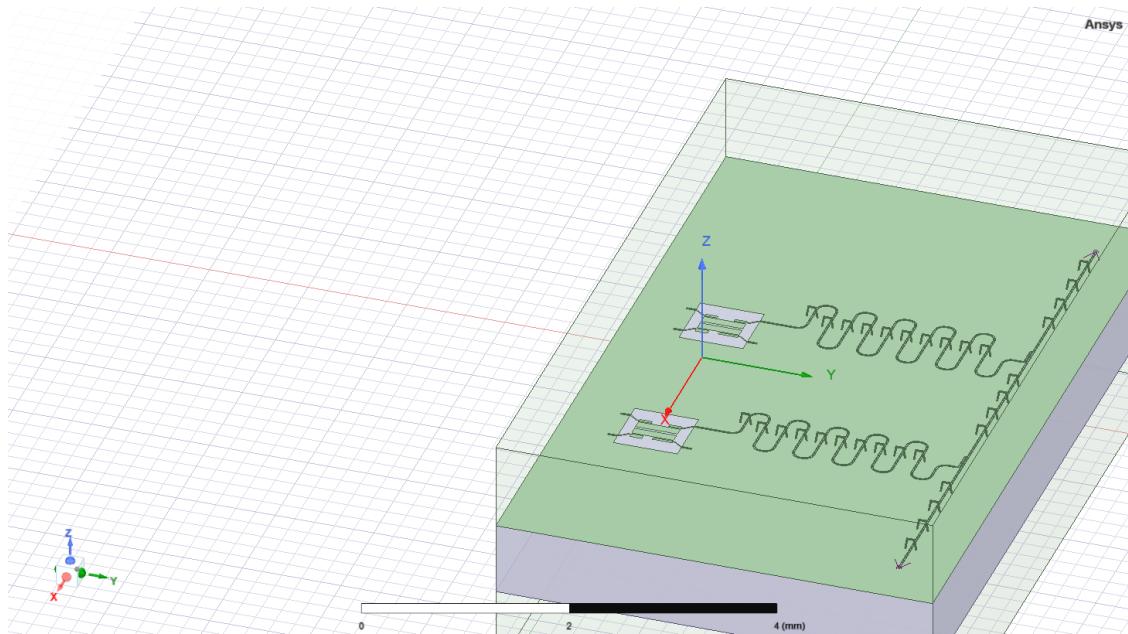
Here, pins `cpw_openRight_end` and `cpw_openLeft_end` are converted into lumped ports with an impedance of 50 Ohms. Neither of the junctions in Q1 or Q2 are rendered.

[15]: `hfss.render_design(selection=[],
 open_pins=[],
 port_list=[('cpw_openRight', 'end', 50),
 ('cpw_openLeft', 'end', 50)],`

```
jj_to_port=[] ,  
ignored_jjs=[('Q1', 'rect_jj'), ('Q2', 'rect_jj')],  
box_plus_buffer = True)
```

(optional) Captures the renderer GUI.

[16]: `hfss.save_screenshot()`



[16]: `WindowsPath('C:/Users/Bartu/Desktop/qiskit-metal/tutorials/4 Analysis/B. Advanced - Direct use of the renderers/ansys.png')`

Create the frequency sweep to observe the impedance, admittance and scattering matrices.

[17]: `hfss.add_sweep(setup_name="Setup",
name="Sweep",
start_ghz=4.0,
stop_ghz=8.0,
count=2001,
type="Interpolating")`

```
INFO 05:02PM [get_setup]:      Opened setup `Setup` (<class  
'pyEPR.ansys.HfssDMSetup'>)
```

[17]: `<pyEPR.ansys.HfssFrequencySweep at 0x29a428ccc70>`

[18]: `hfss.analyze_sweep('Sweep', 'Setup')`

```
INFO 05:02PM [get_setup]:          Opened setup `Setup` (<class
'pyEPR.ansys.HfssDMSetup'>)
INFO 05:02PM [analyze]: Analyzing setup Setup : Sweep
```

Plot S, Y, and Z parameters as a function of frequency. The left and right plots display the magnitude and phase, respectively.

```
[19]: hfss.plot_params(['S11', 'S21'])
```

```
[19]: (      S11           S21
 4.000 -0.158800-0.037349j -0.273572+0.947917j
 4.002 -0.158855-0.037250j -0.272968+0.948086j
 4.004 -0.158910-0.037152j -0.272364+0.948254j
 4.006 -0.158965-0.037053j -0.271760+0.948422j
 4.008 -0.159020-0.036955j -0.271156+0.948589j
 ...
 ...
 7.992 -0.040227+0.090588j  0.832491+0.545099j
 7.994 -0.040121+0.090540j  0.832850+0.544566j
 7.996 -0.040016+0.090492j  0.833209+0.544033j
 7.998 -0.039910+0.090444j  0.833568+0.543499j
 8.000 -0.039805+0.090396j  0.833926+0.542966j

[2001 rows x 2 columns],
<Figure size 1000x600 with 2 Axes>)
```

```
[20]: hfss.plot_params(['Y11', 'Y21'])
```

```
[20]: (      Y11           Y21
 4.000 0.000000-0.007155j  0.000000-0.024731j
 4.002 -0.000000-0.007139j 0.000000-0.024727j
 4.004 -0.000000-0.007122j 0.000000-0.024722j
 4.006 -0.000000-0.007106j 0.000000-0.024717j
 4.008 -0.000000-0.007089j 0.000000-0.024713j
 ...
 ...
 7.992 0.000000+0.036475j -0.000000-0.044228j
 7.994 0.000000+0.036528j -0.000000-0.044273j
 7.996 0.000000+0.036580j -0.000000-0.044318j
 7.998 0.000000+0.036633j -0.000000-0.044363j
 8.000 0.000000+0.036686j -0.000000-0.044409j

[2001 rows x 2 columns],
<Figure size 1000x600 with 2 Axes>)
```

```
[22]: hfss.plot_params(['Z11', 'Z21'])
```

```
[22]: (          Z11          Z21
 4.000  0.000000-11.975315j -0.000000+43.899162j
 4.002 -0.000003-11.945670j  0.000005+43.890849j
 4.004 -0.000006-11.916036j  0.000010+43.882557j
 4.006 -0.000009-11.886413j  0.000015+43.874287j
 4.008 -0.000012-11.856801j  0.000020+43.866038j
 ...
 ...
 7.992  0.000079+66.028031j  0.000030+77.063541j
 7.994  0.000060+66.119666j  0.000023+77.139253j
 7.996  0.000040+66.211481j  0.000015+77.215144j
 7.998  0.000020+66.303476j  0.000008+77.291217j
 8.000 -0.000000+66.395651j -0.00000+077.367470j
```

```
[2001 rows x 2 columns],
<Figure size 1000x600 with 2 Axes>)
```

Finally, disconnect from Ansys.

```
[23]: em1.close()
```

(optional) close the GUI.

```
[24]: gui.main_window.close()
```

```
[24]: True
```

```
[ ]:
```

3.27 Example full chip design

```
[1]: %load_ext autoreload  
%autoreload 2
```

Make sure to have the right kernel selected!

```
[2]: import qiskit_metal as metal  
from qiskit_metal import designs, draw  
from qiskit_metal import MetalGUI, Dict, open_docs  
  
%metal_heading Welcome to Qiskit Metal!
```

```
<IPython.core.display.HTML object>
```

Welcome to Qiskit Metal!

For this example tutorial, we will attempt to create a multi qubit chip with a variety of components. We will want to generate the layout, simulate/analyze and tune the chip to hit the parameters we are wanting, finally rendering to a GDS file.

One could generate subsections of the layout and tune individual components first, but in this case we will create all of the layout. We will be using both transmon pockets and crossmons, meandered and simple transmission lines, capacitive couplers, and launchers for wirebond connections. So we will import these, and also create a design instance and launch the GUI.

3.28 Layout

```
[3]: from qiskit_metal qlibrary qubits transmon_pocket_6 import TransmonPocket6  
from qiskit_metal qlibrary qubits transmon_cross_fl import TransmonCrossFL  
  
from qiskit_metal qlibrary couplers tunable_coupler_01 import TunableCoupler01  
    ↪TunableCoupler01  
  
from qiskit_metal qlibrary tlines meandered import RouteMeander  
from qiskit_metal qlibrary tlines pathfinder import RoutePathfinder  
from qiskit_metal qlibrary tlines anchored_path import RouteAnchors  
  
from qiskit_metal qlibrary lumped cap_n_interdigital import CapNInterdigital  
    ↪CapNInterdigital  
from qiskit_metal qlibrary couplers cap_n_interdigital_tee import CapNInterdigitalTee  
    ↪CapNInterdigitalTee  
from qiskit_metal qlibrary couplers coupled_line_tee import CoupledLineTee
```

```
from qiskit_metal qlibrary terminations launchpad_wb import
    LaunchpadWirebond
from qiskit_metal qlibrary terminations launchpad_wb_coupled import
    LaunchpadWirebondCoupled
```

```
[4]: design = metal.designs.DesignPlanar()
gui = metal.MetalGUI(design)
```

Since we are likely to be making many changes while tuning and modifying our design, we will enable overwriting. We can also check all of the chip properties to see if we want to change the size or any other parameter.

```
[5]: design.overwrite_enabled = True
design.chips.main
```

```
[5]: {'material': 'silicon',
      'layer_start': '0',
      'layer_end': '2048',
      'size': {'center_x': '0.0mm',
                'center_y': '0.0mm',
                'center_z': '0.0mm',
                'size_x': '9mm',
                'size_y': '6mm',
                'size_z': '-750um',
                'sample_holder_top': '890um',
                'sample_holder_bottom': '1650um'}}}
```

```
[6]: design.chips.main.size.size_x = '11mm'
design.chips.main.size.size_y = '9mm'
```

3.28.0.0.1 The Qubits

We will add a collection of qubits. First we will place a transmon pocket with six connection pads. We can see any options the qubit qcomponent has to figure out what we might want to modify when creating the component. This will include the components default options (which the component designer included) as well as renderer options (which are added based on what renderers are present in Metal).

```
[7]: TransmonPocket6.get_template_options(design)
```

```
[7]: {'pos_x': '0.0um',
      'pos_y': '0.0um',
      'orientation': '0.0',
      'chip': 'main',
```

```
'layer': '1',
'connection_pads': {},
'_default_connection_pads': {'pad_gap': '15um',
'pad_width': '125um',
'pad_height': '30um',
'pad_cpw_shift': '0um',
'pad_cpw_extent': '25um',
'cpw_width': '10um',
'cpw_gap': '6um',
'cpw_extend': '100um',
'pocket_extent': '5um',
'pocket_rise': '0um',
'loc_W': '+1',
'loc_H': '+1'},
'pad_gap': '30um',
'inductor_width': '20um',
'pad_width': '455um',
'pad_height': '90um',
'pocket_width': '650um',
'pocket_height': '650um',
'hfss_wire_bonds': False,
'q3d_wire_bonds': False,
'hfss_inductance': '10nH',
'hfss_capacitance': 0,
'hfss_resistance': 0,
'hfss_mesh_kw_jj': 7e-06,
'q3d_inductance': '10nH',
'q3d_capacitance': 0,
'q3d_resistance': 0,
'q3d_mesh_kw_jj': 7e-06,
'gds_cell_name': 'my_other_junction'}
```

```
[8]: options = dict(
    pad_width = '425 um',
    pocket_height = '650um',
    connection_pads=dict(
        readout = dict(loc_W=0, loc_H=-1, pad_width = '80um', pad_gap = '50um'),
        bus_01 = dict(loc_W=-1, loc_H=-1, pad_width = '60um', pad_gap = '10um'),
        bus_02 = dict(loc_W=-1, loc_H=+1, pad_width = '60um', pad_gap = '10um'),
        bus_03 = dict(loc_W=0, loc_H=+1, pad_width = '90um', pad_gap = '30um'),
```

```

        bus_04 = dict(loc_W=+1, loc_H=+1, pad_width = '60um', pad_gap = '10um'),
        bus_05 = dict(loc_W=+1, loc_H=-1, pad_width = '60um', pad_gap = '10um')
    ))
}

q_main = TransmonPocket6(design, 'Q_Main', options = dict(
    pos_x='0mm',
    pos_y='1mm',
    gds_cell_name = 'FakeJunction_01',
    hfss_inductance = '14nH',
    **options))

gui.rebuild()
gui.autoscale()

```

We then will add a mixture of additional qubits. This is not (though do not let me stop any experimental investigation) a design one would normally create for any experiment of computational purpose, but allows for having a mixture of different components on one chip.

[9] : `TransmonCrossFL.get_template_options(design)`

```

[9]: {'pos_x': '0.0um',
      'pos_y': '0.0um',
      'orientation': '0.0',
      'chip': 'main',
      'layer': '1',
      'connection_pads': {},
      '_default_connection_pads': {'connector_type': '0',
                                    'claw_length': '30um',
                                    'ground_spacing': '5um',
                                    'claw_width': '10um',
                                    'claw_gap': '6um',
                                    'connector_location': '0'},
      'cross_width': '20um',
      'cross_length': '200um',
      'cross_gap': '20um',
      'make_fl': True,
      'fl_options': {'t_top': '15um',
                    't_offset': '0um',
                    't_inductive_gap': '3um',
                    't_width': '5um',
                    't_gap': '3um'},
      'hfss_wire_bonds': False,
      }

```

```
'q3d_wire_bonds': False,
'hfss_inductance': '10nH',
'hfss_capacitance': 0,
'hfss_resistance': 0,
'hfss_mesh_kw_jj': 7e-06,
'q3d_inductance': '10nH',
'q3d_capacitance': 0,
'q3d_resistance': 0,
'q3d_mesh_kw_jj': 7e-06,
'gds_cell_name': 'my_other_junction'}
```

We will add two crossmons with flux lines to the west side of the chip, which we will couple to each other using a tunable coupler. To make sure the various readout and control lines will have space to connect to launchers at the chip edge, we have to be mindful of where we place them, and making sure we have enough space for routing while avoiding cross talk.

```
[10]: Q1 = TransmonCrossFL(design, 'Q1', options = dict(pos_x = '-2.75mm', □
→pos_y='-1.8mm',
connection_pads = dict(
bus_01 = □
→dict(connector_location = '180',claw_length ='95um'),
readout = □
→dict(connector_location = '0')), □
fl_options = dict()))

Q2 = TransmonCrossFL(design, 'Q2', options = dict(pos_x = '-2.75mm', □
→pos_y='-1.2mm', orientation = '180',
connection_pads = dict(
bus_02 = □
→dict(connector_location = '0',claw_length ='95um'),
readout = □
→dict(connector_location = '180')), □
fl_options = dict()))

tune_c_Q12 = TunableCoupler01(design,'Tune_C_Q12', options = dict(pos_x = □
→'-2.81mm', pos_y = '-1.5mm',
□
→orientation=90, c_width='500um'))

gui.rebuild()
gui.autoscale()
```

We then will add three transmon pockets to the north side of the chip, with the intention of

having them in a linear series of coupling to each other, as well as the ‘main’ qubit to the south.

```
[11]: Q3 = TransmonPocket6(design, 'Q3', options = dict(
    pos_x=' -3mm',
    pos_y=' 0.5mm',
    gds_cell_name = 'FakeJunction_01',
    hfss_inductance = '14nH',
    connection_pads = dict(
        bus_03 = dict(loc_W=0, loc_H=-1, pad_width = '80um', pad_gap
        ↵= '15um'),
        bus_q3_q4 = dict(loc_W=1, loc_H=-1, pad_width = '80um', ↵
        ↵pad_gap = '15um'),
        readout = dict(loc_W=0, loc_H=1, pad_width = '80um', pad_gap
        ↵= '50um')))

Q4 = TransmonPocket6(design, 'Q4', options = dict(
    pos_x=' 0mm',
    pos_y=' 1mm',
    gds_cell_name = 'FakeJunction_01',
    hfss_inductance = '14nH',
    connection_pads = dict(
        bus_04 = dict(loc_W=0, loc_H=-1, pad_width = '80um', pad_gap
        ↵= '15um'),
        bus_q3_q4 = dict(loc_W=-1, loc_H=-1, pad_width = '80um', ↵
        ↵pad_gap = '15um'),
        bus_q4_q5 = dict(loc_W=1, loc_H=-1, pad_width = '80um', ↵
        ↵pad_gap = '15um'),
        readout = dict(loc_W=0, loc_H=1, pad_width = '80um', pad_gap
        ↵= '50um')))

Q5 = TransmonPocket6(design, 'Q5', options = dict(
    pos_x=' 3mm',
    pos_y=' 0.5mm',
    gds_cell_name = 'FakeJunction_01',
    hfss_inductance = '14nH',
    connection_pads = dict(
        bus_05 = dict(loc_W=0, loc_H=-1, pad_width = '80um', pad_gap
        ↵= '15um'),
        bus_q4_q5 = dict(loc_W=-1, loc_H=-1, pad_width = '80um', ↵
        ↵pad_gap = '15um'),
        readout = dict(loc_W=0, loc_H=1, pad_width = '80um', pad_gap
        ↵= '50um')))
```

3.28.0.0.2 The Busses

We now couple the qubits to each other, primarily using RouteMeander. Although one needs to run simulations to properly tune the line lengths for target frequencies, an initial estimate could be determined from the below method;

```
[12]: from qiskit_metal.analyses.em.cpw_calculations import guided_wavelength

def find_resonator_length(frequency, line_width, line_gap, N):
    #frequency in GHz
    #line_width/line_gap in um
    #N -> 2 for lambda/2, 4 for lambda/4

    [lambdaG, etfSqrt, q] = guided_wavelength(frequency*10**9,
    ↪line_width*10**-6,
                                         line_gap*10**-6,
    ↪750*10**-6, 200*10**-9)
    return str(lambdaG/N*10**3)+" mm"
```

As we are not worried about creating a functional chip in this tutorial, we will give the resonators somewhat arbitrary lengths. First coupling the two crossmons to Q_Main.

```
[13]: bus_01 = RouteMeander(design, 'Bus_01', options = dict(hfss_wire_bonds = True,
                                                               pin_inputs=Dict(
                                                                 start_pin=Dict(
                                                                   component='Q_Main',
                                                                   pin='bus_01'),
                                                                 end_pin=Dict(
                                                                   component='Q1',
                                                                   pin='bus_01')
                                                               ),
                                                               lead=Dict(
                                                                 start_straight='125um',
                                                                 end_straight = '225um'
                                                               ),
                                                               meander=Dict(
                                                                 asymmetry = '1305um'),
                                                               fillet = "99um",
                                                               total_length = '6mm')))

bus_02 = RouteMeander(design, 'Bus_02', options = dict(hfss_wire_bonds = True,
                                                               pin_inputs=Dict(
                                                                 start_pin=Dict(
```

```

        component='Q_Main',
        pin='bus_02'),
end_pin=Dict(
    component='Q2',
    pin='bus_02')
),
lead=Dict(
    start_straight='325um',
    end_straight = '125um'
),
meander=Dict(
    asymmetry = '450um'),
fillet = "99um",
total_length = '6.4mm'))
gui.rebuild()

```

Then the three transmon pockets on the north side to Q_Main.

```
[14]: bus_03 = RouteMeander(design, 'Bus_03', options = dict(hfss_wire_bonds = True,
pin_inputs=Dict(
    start_pin=Dict(
        component='Q_Main',
        pin='bus_03'),
end_pin=Dict(
    component='Q3',
    pin='bus_03')
),
lead=Dict(
    start_straight='225um',
    end_straight = '25um'
),
meander=Dict(
    asymmetry = '50um'),
fillet = "99um",
total_length = '6.8mm'))

#To help set the right spacing, jogs can be used to set some initially controlled routing paths
from collections import OrderedDict
jogs_start = OrderedDict()
jogs_start[0] = ["L", '250um']
```

```
jogs_start[1] = ["R", '200um']

jogs_end = OrderedDict()
jogs_end[0] = ["L", '600um']

bus_04 = RouteMeander(design, 'Bus_04', options = dict(hfss_wire_bonds = False,
→True,
    pin_inputs=Dict(
        start_pin=Dict(
            component='Q_Main',
            pin='bus_04'),
        end_pin=Dict(
            component='Q4',
            pin='bus_04')
    ),
    lead=Dict(
        start_straight='225um',
        #end_straight = '25um',
        ),
    →start_jogged_extension=jogs_start,
        #end_jogged_extension = False,
    →jogs_end
    ),
    meander=Dict(
        asymmetry = '150um'),
    fillet = "99um",
    total_length = '7.2mm'))

bus_05 = RouteMeander(design, 'Bus_05', options = dict(hfss_wire_bonds = False,
→True,
    pin_inputs=Dict(
        start_pin=Dict(
            component='Q_Main',
            pin='bus_05'),
        end_pin=Dict(
            component='Q5',
            pin='bus_05')
    ),
    lead=Dict(
        start_straight='225um',
        end_straight = '25um'
    ),
    meander=Dict(
        asymmetry = '50um'),
```

```

        fillet = "99um",
        total_length = '7.6mm')))

gui.rebuild()

```

Finally the three transmon pockets on the north side to each other. This concludes the interconnectivity between the qubits.

```

[15]: bus_q3_q4 = RouteMeander(design, 'Bus_Q3_Q4', options = dict(
    hfss_wire_bonds = True,
    pin_inputs=Dict(
        start_pin=Dict(
            component='Q3',
            pin='bus_q3_q4'),
        end_pin=Dict(
            component='Q4',
            pin='bus_q3_q4')
    ),
    lead=Dict(
        start_straight='125um',
        end_straight = '125um'
    ),
    meander=Dict(
        asymmetry = '50um'),
    fillet = "99um",
    total_length = '6.4mm'))

bus_q4_q5 = RouteMeander(design, 'Bus_Q4_Q5', options = dict(
    hfss_wire_bonds = True,
    pin_inputs=Dict(
        start_pin=Dict(
            component='Q4',
            pin='bus_q4_q5'),
        end_pin=Dict(
            component='Q5',
            pin='bus_q4_q5')
    ),
    lead=Dict(
        start_straight='125um',
        end_straight = '25um'
    ),
    meander=Dict(
        asymmetry = '50um'),
    fillet = "99um",

```

```

    total_length = '6.8mm'))
gui.rebuild()

```

3.28.0.0.3 The Readouts and Control Lines

The intention for this design is to have the three north transmon pockets be multiplexed to one readout line. The crossmons to their own readouts, as well as Q_Main. The tunable coupler, and the two crossmons also have flux lines which need to be connected to launchers. First we will place the wirebond launchers at the edges of the chip.

```

[16]: launch_qmain_read = LaunchpadWirebond(design, 'Launch_QMain_Read',
                                          options = dict(pos_x = '2mm', pos_y = '-4mm', orientation = '90'))

launch_q1_fl = LaunchpadWirebond(design, 'Launch_Q1_FL', options =
                                  dict(pos_x = '0mm', pos_y = '-4mm', orientation = '90',
                                       trace_width = '5um',
                                       trace_gap = '3um',))

launch_q1_read = LaunchpadWirebondCoupled(design, 'Launch_Q1_Read',
                                           options = dict(pos_x = '-2mm', pos_y = '-4mm', orientation = '90'))

launch_tcoup_fl = LaunchpadWirebond(design, 'Launch_TuneC_FL', options =
                                      dict(pos_x = '-4mm', pos_y = '-4mm', orientation = '90',
                                           trace_width = '5um',
                                           trace_gap = '3um',))

launch_tcoup_read = LaunchpadWirebondCoupled(design, 'Launch_TuneC_Read',
                                              options = dict(pos_x = '-5mm', pos_y = '-3mm', orientation = '0'))

launch_q2_read = LaunchpadWirebondCoupled(design, 'Launch_Q2_Read',
                                           options = dict(pos_x = '-5mm', pos_y = '-1mm', orientation = '0'))
launch_q2_fl = LaunchpadWirebond(design, 'Launch_Q2_FL', options =
                                  dict(pos_x = '-5mm', pos_y = '1mm', orientation = '0',
                                       trace_width = '5um',
                                       trace_gap = '3um',))

launch_nw = LaunchpadWirebond(design, 'Launch_NW', options = dict(pos_x =
                                                               '-5mm', pos_y = '3mm', orientation=0))

```



```
launch_ne = LaunchpadWirebond(design, 'Launch_NE', options = dict(pos_x = 5, pos_y = 3, orientation=180))

gui.rebuild()
```

We then will add in the readout resonators for Q_Main, Q1, Q2 and the tuneable coupler. We will add a finger capacitor for the Q_Main readout, instead of just using the LaunchpadWirebondCoupled.

[17] : #Main Readout

```

read_q_main_cap = CapNInterdigital(design, 'Read_Q_Main_Cap', options =
    ↵dict(pos_x = '2mm', pos_y = '-3.5mm', orientation = '0'))

jogs_end = OrderedDict()
jogs_end[0] = ["L", '600um']

jogs_start = OrderedDict()
jogs_start[0] = ["L", '250um']

read_q_main = RouteMeander(design, 'Read_Q_Main', options =
    ↵dict(hfss_wire_bonds = True,
          pin_inputs=Dict(
              start_pin=Dict(
                  component='Q_Main',
                  pin='readout'),
              end_pin=Dict(
                  component='Read_Q_Main_Cap',
                  pin='north_end')
          ),
          lead=Dict(
              start_straight='725um',
              end_straight = '625um',
              start_jogged_extension =
    ↵jogs_start,
              end_jogged_extension =
    ↵jogs_end
          ),
          meander=Dict(
              asymmetry = '50um'),
          fillet = "99um",
          )
      )

```

```

        total_length = '5.6mm'))
read_q_main_cap_launch = RoutePathfinder(design,
    ↪'Read_Q_Main_Cap_Launch', options = dict(hfss_wire_bonds = True,
                                                pin_inputs = dict(
                                                    start_pin=Dict(
                                                        ↣
                                                    ↪component='Read_Q_Main_Cap',
                                                    pin='south_end'),
                                                    end_pin=Dict(
                                                        ↣
                                                    ↪component='Launch_QMain_Read',
                                                    pin='tie')),
                                                lead=Dict(
                                                    start_straight='0um',
                                                    end_straight = '0um',
#start_jogged_extension = ↣
                                                    ↪jogs_start,
#end_jogged_extension = ↣
                                                    ↪jogs_end
                                                )))

gui.rebuild()

```

[18] : #Crossmon's Readouts

```

jogs_end = OrderedDict()
jogs_end[0] = ["L", '600um']

jogs_start = OrderedDict()
jogs_start[0] = ["L", '250um']

read_q1 = RouteMeander(design, 'Read_Q1', options = dict(hfss_wire_bonds = ↣
    ↪True,
                                                pin_inputs=Dict(
                                                    start_pin=Dict(
                                                        component='Q1',
                                                        pin='readout'),
                                                    end_pin=Dict(
                                                        ↣
                                                    ↪component='Launch_Q1_Read',

```

```

                pin='tie')
            ),
            lead=Dict(
                start_straight='250um',
                end_straight = '25um',
                #start_jogged_extension =_
→jogs_start,
                #end_jogged_extension =_
→jogs_end
            ),
            meander=Dict(
                asymmetry = '50um'),
                fillet = "99um",
                total_length = '6.8mm'))
        )

jogs_end = OrderedDict()
jogs_end[0] = ["L", '600um']

jogs_start = OrderedDict()
jogs_start[0] = ["L", '250um']

read_tunec = RouteMeander(design, 'Read_TuneC', options =_
→dict(hfss_wire_bonds = True,
pin_inputs=Dict(
    start_pin=Dict(
        pin='Control'),
    end_pin=Dict(
        pin='Control'),
        ),
lead=Dict(
    start_straight='1525um',
    end_straight = '125um',
    #start_jogged_extension =_
→jogs_start,
                #end_jogged_extension =_
→jogs_end
            ),
            meander=Dict(

```

```

        asymmetry = '50um'),
fillet = "99um",
total_length = '5.8mm')))

jogs_end = OrderedDict()
jogs_end[0] = ["L", '600um']

jogs_start = OrderedDict()
jogs_start[0] = ["L", '250um']

read_q2 = RouteMeander(design, 'Read_Q2', options = dict(hfss_wire_bonds = False,
→True,
    pin_inputs=Dict(
        start_pin=Dict(
            component='Q2',
            pin='readout'),
        end_pin=Dict(
            □
→component='Launch_Q2_Read',
            pin='tie')
    ),
    lead=Dict(
        start_straight='350um',
        end_straight = '0um',
        #start_jogged_extension = False,
→jogs_start,
        #end_jogged_extension = False
→jogs_end
    ),
    meander=Dict(
        asymmetry = '-450um'),
        fillet = "99um",
        total_length = '5.4mm'))
)

gui.rebuild()

```

Finishing off this section of the chip by connecting the flux lines to appropriate wirebond launch pads.

[19]: #Crossmon flux lines

```

flux_line_Q1 = RoutePathfinder(design, 'Flux_Line_Q1', options =_
    ↪dict(hfss_wire_bonds = True,
          pin_inputs=Dict(
              start_pin=Dict(
                  component='Q1',
                  pin='flux_line'),
              end_pin=Dict(
                  □
                  component='Launch_Q1_FL',
                  pin='tie'))),
          fillet = '99um',
          trace_width = '5um',
          trace_gap = '3um',
          #anchors = anchors
    ))
)

jogs_start = OrderedDict()
jogs_start[0] = ["L", '750um']

flux_line_tunec = RoutePathfinder(design, 'Flux_Line_TuneC', options =_
    ↪dict(hfss_wire_bonds = True,
          pin_inputs=Dict(
              start_pin=Dict(
                  □
                  component='Tune_C_Q12',
                  pin='Flux'),
              end_pin=Dict(
                  □
                  component='Launch_TuneC_FL',
                  pin='tie'))),
          lead=Dict(
              start_straight='875um',
              end_straight = '350um',
              start_jogged_extension =_
    ↪jogs_start,
                  #end_jogged_extension =_
    ↪jogs_end
            ),
          fillet = '99um',
          trace_width = '5um',
          trace_gap = '3um',
          #anchors = anchors
    ))
)

```

```

        )))

jogs_start = OrderedDict()
jogs_start[0] = ["L", '525um']
jogs_start[1] = ["R", '625um']

flux_line_Q2 = RoutePathfinder(design, 'Flux_Line_Q2', options = dict(
    hfss_wire_bonds = True,
    pin_inputs=Dict(
        start_pin=Dict(
            component='Q2',
            pin='flux_line'),
        end_pin=Dict(
            component='Launch_Q2_FL',
            pin='tie'))),
    lead=Dict(
        start_straight='175um',
        end_straight = '150um',
        start_jogged_extension = dict(
            jogs_start,
            #end_jogged_extension = dict(
                jogs_end
            )),
        fillet = '99um',
        trace_width = '5um',
        trace_gap = '3um',
        #anchors = anchors
    )))
gui.rebuild()

```

Shifting our focus now to the three transmon pockets in the north. As we want these to be multiplexed to a single readout line, we will add in a few three port components, such as the CoupledLineTee and CapNInterdigitalTee. Q3 will have an inductive coupling to the readout line (as we want a lambda/4 resonator), Q4 will have a simple gap capacitor, and Q5 will have an interdigitated capacitor.

```
[20]: q3_read_T = CoupledLineTee(design, 'Q3_Read_T', options=dict(pos_x = '-3mm', pos_y = '3mm',
    orientation = '0',
    coupling_length = '200um',
    ))

```

```

        open_termination = open_termination
        ↵= False))
#We use finger count to set the width of the gap capacitance, ->
    ↵N*cap_width + (N-1)*cap_gap
q4_read_T = CapNInterdigitalTee(design, 'Q4_Read_T', options=dict(pos_x =
    ↵'0mm', pos_y = '3mm',
                                         orientation = orientation
    ↵'0',
                                         finger_length = finger_length
    ↵= '0um',
                                         finger_count = finger_count
    ↵'8'))
q5_read_T = CapNInterdigitalTee(design, 'Q5_Read_T', options=dict(pos_x =
    ↵'3mm', pos_y = '3mm',
                                         orientation = orientation
    ↵'0',
                                         finger_length = finger_length
    ↵= '50um',
                                         finger_count = finger_count
    ↵'11'))
gui.rebuild()

```

We add in the readout resonators to each respective qubit.

```
[21]: read_q3 = RouteMeander(design, 'Read_Q3', options = dict(hfss_wire_bonds =
    ↵True,
                                         pin_inputs=Dict(
                                             start_pin=Dict(
                                                 component='Q3',
                                                 pin='readout'),
                                             end_pin=Dict(
                                                 component='Q3_Read_T',
                                                 pin='second_end')
                                         ),
                                         lead=Dict(
                                             start_straight='150um',
                                             end_straight = '150um',
                                             #start_jogged_extension =
    ↵jogs_start,
                                         #end_jogged_extension =
    ↵jogs_end
                                         )),
                                         )
```

```

meander=Dict(
    asymmetry = '0um'),
fillet = "99um",
total_length = '5mm'))

read_q4 = RouteMeander(design, 'Read_Q4', options = dict(hfss_wire_bonds = False,
→True,
    pin_inputs=Dict(
        start_pin=Dict(
            component='Q4',
            pin='readout'),
        end_pin=Dict(
            component='Q4_Read_T',
            pin='second_end')
    ),
    lead=Dict(
        start_straight='125um',
        end_straight = '125um',
        #start_jogged_extension = False,
→jogs_start,
        #end_jogged_extension = False,
→jogs_end
    ),
    meander=Dict(
        asymmetry = '0um'),
    fillet = "99um",
    total_length = '5.8mm'))

read_q5 = RouteMeander(design, 'Read_Q5', options = dict(hfss_wire_bonds = False,
→True,
    pin_inputs=Dict(
        start_pin=Dict(
            component='Q5',
            pin='readout'),
        end_pin=Dict(
            component='Q5_Read_T',
            pin='second_end')
    ),
    lead=Dict(
        start_straight='125um',
        end_straight = '125um',
        #start_jogged_extension = False,
→jogs_start,
        #end_jogged_extension = False,
→jogs_end,
        #start_jogged_extension = False,
→jogs_end
    ),
    meander=Dict(
        asymmetry = '0um'),
    fillet = "99um",
    total_length = '5.8mm'))

```

```

        #end_jogged_extension = □
→ jogs_end
),
meander=Dict(
    asymmetry = '0um',
fillet = "99um",
total_length = '5.4mm'))
gui.rebuild()

```

We complete the layout by connecting the multiplexed readout line to the launchpads on either side of the chip.

```

[22]: mp_tl_01 = RoutePathfinder(design, 'ML_TL_01', options = □
→ dict(hfss_wire_bonds = True,
       pin_inputs = dict(
           start_pin=Dict(
               □
→ component='Launch_NW',
               pin='tie'),
           end_pin=Dict(
               □
→ component='Q3_Read_T',
               □
→ pin='prime_start'))
       )))

mp_tl_02 = RoutePathfinder(design, 'ML_TL_02', options = □
→ dict(hfss_wire_bonds = True,
       pin_inputs = dict(
           start_pin=Dict(
               □
→ component='Q3_Read_T',
               pin='prime_end'),
           end_pin=Dict(
               □
→ component='Q4_Read_T',
               □
→ pin='prime_start'))
       )))

mp_tl_03 = RoutePathfinder(design, 'ML_TL_03', options = □
→ dict(hfss_wire_bonds = True,
       pin_inputs = dict(

```

```

start_pin=Dict(
    □
    ↵component='Q4_Read_T',
        pin='prime_end'),
end_pin=Dict(
    □
    ↵component='Q5_Read_T',
        □
    ↵pin='prime_start'))
    □

mp_tl_04 = RoutePathfinder(design, 'ML_TL_04', options =□
    ↵dict(hfss_wire_bonds = True,
          pin_inputs = dict(
            start_pin=Dict(
                □
                ↵component='Q5_Read_T',
                    pin='prime_end'),
            end_pin=Dict(
                □
                ↵component='Launch_NE',
                    pin='tie')))
    □

gui.rebuild()

```

With this, we have completed the construction of our layout.

Now, anyone familiar with chip design might find some of the location choices to be sub-optimal, with large sections of your chip left unused, or perhaps some CPW transmission lines running a bit closer to each other than would be ideal for avoiding cross talk concerns. These could be address by shifting the origin of your chip, or modifying component options to better compact your layout and alleviate crosstalk concerns.

For this tutorial, we aren't too concerned how much space we may use up on our fictional chip, so we will instead continue on to analysis and tuning.

3.29 Analyze

3.29.1 Capacitance Extraction and LOM

First we want to quickly look at the qubit parameters. Initial simulation and analysis is to use a lumped element approximation, by extracting the capacitance matrix of the qubit. We first analyze the qubit `Q_Main`, by first extracting the matrix and then using LOM analysis on it.

For starters, let's select the appropriate QAnalysis class.

```
[23]: from qiskit_metal.analyses.quantization import LOManalysis  
c1 = LOManalysis(design, "q3d")
```

We can check if we wish to change any of the default options for the analysis. You should modify the number of passes and convergence based on the accuracy you require for your simulation.

Depending on the complexity of the simulation, it could take a minute, or multiple hours. It is generally best to start with a small number of `max_passes` if you are unsure, so can you get a sense on the timing. As each adaptive pass adds additional tetrahedrons, the simulation time per pass will increase significantly (as well as the amount of system memory necessary).

```
[24]: c1.sim.setup
```

```
[24]: {'name': 'Setup',  
       'reuse_selected_design': True,  
       'reuse_setup': True,  
       'freq_ghz': 5.0,  
       'save_fields': False,  
       'enabled': True,  
       'max_passes': 15,  
       'min_passes': 2,  
       'min_converged_passes': 2,  
       'percent_error': 0.5,  
       'percent_refinement': 30,  
       'auto_increase_solution_order': True,  
       'solution_order': 'High',  
       'solver_type': 'Iterative'}
```

```
[25]: c1.sim.setup.name = 'Tune_Q_Main'  
c1.sim.setup.max_passes = 16  
c1.sim.setup.min_converged_passes = 2  
c1.sim.setup.percent_error = 0.05  
c1.sim.setup
```

```
[25]: {'name': 'Tune_Q_Main',  
       'reuse_selected_design': True,  
       'reuse_setup': True,  
       'freq_ghz': 5.0,  
       'save_fields': False,  
       'enabled': True,  
       'max_passes': 16,
```

```
'min_passes': 2,
'min_converged_passes': 2,
'percent_error': 0.05,
'percent_refinement': 30,
'auto_increase_solution_order': True,
'solution_order': 'High',
'solver_type': 'Iterative'}
```

Next we will want to run the simulation for Q_Main. To obtain the complete capacitance matrix from this simulation, we will want to terminate the unconnected pins of Q_Main with opens, so that they are regarded as isolated charge islands in the simulation. You will need to list all of the pin names in the call. To recall them, one can look at the GUI, or check the pin dictionary attached to Q_Main: q_main.pins.keys().

```
[26]: c1.sim.run(name="Q_Main", components=['Q_Main'],
    ↪open_terminations=[('Q_Main', 'readout'), ('Q_Main', ↪
    ↪'bus_01'), ('Q_Main', 'bus_02'), ('Q_Main', 'bus_03'),
    ('Q_Main', 'bus_04'), ('Q_Main', ↪
    ↪'bus_05')])
```

```
INFO 09:35PM [connect_project]: Connecting to Ansys Desktop API...
INFO 09:35PM [load_ansys_project]:      Opened Ansys App
INFO 09:35PM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0
INFO 09:35PM [load_ansys_project]:      Opened Ansys Project
    Folder:   C:/Users/Bartu/Documents/Ansoft/
    Project:  Project11
INFO 09:35PM [connect_design]: No active design found (or error getting ↪
    ↪active
design).
INFO 09:35PM [connect]:           Connected to project "Project11". No design
detected
INFO 09:35PM [connect_design]:  Opened active design
    Design:   Q_Main_q3d [Solution type: Q3D]
WARNING 09:35PM [connect_setup]:      No design setup detected.
WARNING 09:35PM [connect_setup]:      Creating Q3D default setup.
INFO 09:35PM [get_setup]:      Opened setup `Setup` (<class
    'pyEPR.ansys.AnsysQ3DSetup'&)
INFO 09:35PM [get_setup]:      Opened setup `Tune_Q_Main` (<class
    'pyEPR.ansys.AnsysQ3DSetup'&)
INFO 09:35PM [analyze]: Analyzing setup Tune_Q_Main
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpv1vx9vh0.txt, C, ,
Tune_Q_Main:LastAdaptive, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 1, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
```

```
(C:\Users\Bartu\AppData\Local\Temp\tmpb_rz8bc1.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 1, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpl2atd80u.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 2, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp3zriygdb.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 3, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpm5bffvao.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 4, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmps0eiad8w.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 5, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpvt359htx.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 6, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpbk76j9of.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 7, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp5nz08tj3.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 8, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp69ib3puu.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 9, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpmttxats5.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 10, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp1s_axrgs.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 11, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmp5q4pgywu.txt, C, ,
```

```
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 12, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpxltl1xl0.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 13, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpg26vkpnbt.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 14, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpk7go0k0s.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 15, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpcls_vq53.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 16, False
INFO 09:38PM [get_matrix]: Exporting matrix data to
(C:\Users\Bartu\AppData\Local\Temp\tmpndd2xp0h.txt, C, ,
Tune_Q_Main:AdaptivePass, "Original", "ohm", "nH", "fF", "mSie", 5000000000,
Maxwell, 17, False
```

With the simulation completed, we can look at the capacitance matrix.

[27] : c1.sim.capacitance_matrix

[27] :	bus_01_connector_pad_Q_Main \
bus_01_connector_pad_Q_Main	41.02202
bus_02_connector_pad_Q_Main	-0.24001
bus_03_connector_pad_Q_Main	-0.11718
bus_04_connector_pad_Q_Main	-0.04478
bus_05_connector_pad_Q_Main	-0.07871
ground_main_plane	-29.00926
pad_bot_Q_Main	-9.27019
pad_top_Q_Main	-1.06826
readout_connector_pad_Q_Main	-0.69727
	bus_02_connector_pad_Q_Main \
bus_01_connector_pad_Q_Main	-0.24001
bus_02_connector_pad_Q_Main	41.01042
bus_03_connector_pad_Q_Main	-0.75925
bus_04_connector_pad_Q_Main	-0.07821
bus_05_connector_pad_Q_Main	-0.04477
ground_main_plane	-28.99122

pad_bot_Q_Main	-1.07401	
pad_top_Q_Main	-9.22018	
readout_connector_pad_Q_Main	-0.10641	
bus_03_connector_pad_Q_Main	\	
bus_01_connector_pad_Q_Main	-0.11718	
bus_02_connector_pad_Q_Main	-0.75925	
bus_03_connector_pad_Q_Main	45.51027	
bus_04_connector_pad_Q_Main	-0.75924	
bus_05_connector_pad_Q_Main	-0.11711	
ground_main_plane	-31.96340	
pad_bot_Q_Main	-1.30844	
pad_top_Q_Main	-9.60708	
readout_connector_pad_Q_Main	-0.15778	
bus_04_connector_pad_Q_Main	\	
bus_01_connector_pad_Q_Main	-0.04478	
bus_02_connector_pad_Q_Main	-0.07821	
bus_03_connector_pad_Q_Main	-0.75924	
bus_04_connector_pad_Q_Main	41.02591	
bus_05_connector_pad_Q_Main	-0.24007	
ground_main_plane	-29.00786	
pad_bot_Q_Main	-1.07386	
pad_top_Q_Main	-9.21951	
readout_connector_pad_Q_Main	-0.10633	
bus_05_connector_pad_Q_Main	✉	
→ground_main_plane \		
bus_01_connector_pad_Q_Main	-0.07871	-29.
→00926		
bus_02_connector_pad_Q_Main	-0.04477	-28.
→99122		
bus_03_connector_pad_Q_Main	-0.11711	-31.
→96340		
bus_04_connector_pad_Q_Main	-0.24007	-29.
→00786		
bus_05_connector_pad_Q_Main	41.00676	-28.
→99280		
ground_main_plane	-28.99280	308.
→86196		
pad_bot_Q_Main	-9.27031	-33.
→31480		

pad_top_Q_Main	-1.06824	-32.
→78455		
readout_connector_pad_Q_Main	-0.69721	-31.
→48043		
bus_01_connector_pad_Q_Main	pad_bot_Q_Main	pad_top_Q_Main \
	-9.27019	-1.06826
bus_02_connector_pad_Q_Main	-1.07401	-9.22018
bus_03_connector_pad_Q_Main	-1.30844	-9.60708
bus_04_connector_pad_Q_Main	-1.07386	-9.21951
bus_05_connector_pad_Q_Main	-9.27031	-1.06824
ground_main_plane	-33.31480	-32.78455
pad_bot_Q_Main	95.77815	-31.23769
pad_top_Q_Main	-31.23769	97.47685
readout_connector_pad_Q_Main	-7.05118	-1.13128
	readout_connector_pad_Q_Main	
bus_01_connector_pad_Q_Main		-0.69727
bus_02_connector_pad_Q_Main		-0.10641
bus_03_connector_pad_Q_Main		-0.15778
bus_04_connector_pad_Q_Main		-0.10633
bus_05_connector_pad_Q_Main		-0.69721
ground_main_plane		-31.48043
pad_bot_Q_Main		-7.05118
pad_top_Q_Main		-1.13128
readout_connector_pad_Q_Main		42.10850

But more importantly, we can use that matrix to run LOM analysis.

```
[28]: c1.setup.junctions = Dict({'Lj': 14, 'Cj': 2})
c1.setup.freq_readout = 7.0
c1.setup.freq_bus = [5.6, 5.7, 5.8, 5.9, 6.0] # list of the bus
    →frequencies

c1.run_lom()
c1.lumped_oscillator_all
```

[6, 7] [8 0 1 2 3 4]

Predicted Values

Transmon Properties
 f_Q 4.963350 [GHz]
 EC 298.705685 [MHz]
 EJ 11.671114 [GHz]
 α -351.429802 [MHz]

dispersion 85.686985 [KHz]
Lq 13.994355 [nH]
Cq 64.847203 [fF]
T1 139.295106 [us]

****Coupling Properties****

tCqbus1 3.016122 [fF]
gbus1_in_MHz 43.824651 [MHz]
 χ_{-bus1} -0.287267 [MHz]
1/T1bus1 223.154777 [Hz]
T1bus1 713.204285 [us]

tCqbus2 4.171938 [fF]
gbus2_in_MHz 48.509816 [MHz]
 χ_{-bus2} -2.644600 [MHz]
1/T1bus2 287.112296 [Hz]
T1bus2 554.329945 [us]

tCqbus3 -4.002416 [fF]
gbus3_in_MHz -47.367041 [MHz]
 χ_{-bus3} -1.981769 [MHz]
1/T1bus3 204.233533 [Hz]
T1bus3 779.279196 [us]

tCqbus4 -4.074385 [fF]
gbus4_in_MHz -49.045291 [MHz]
 χ_{-bus4} -1.715966 [MHz]
1/T1bus4 169.531149 [Hz]
T1bus4 938.794691 [us]

tCqbus5 -4.002161 [fF]
gbus5_in_MHz -49.020763 [MHz]
 χ_{-bus5} -1.414727 [MHz]
1/T1bus5 134.939593 [Hz]
T1bus5 1179.453265 [us]

tCqbus6 4.172009 [fF]
gbus6_in_MHz 51.964727 [MHz]
 χ_{-bus6} -1.335296 [MHz]
1/T1bus6 123.602483 [Hz]
T1bus6 1287.635485 [us]
Bus-Bus Couplings
gbus1_2 3.325304 [MHz]
gbus1_3 1.913332 [MHz]

```
gbus1_4 2.174245 [MHz]
gbus1_5 1.979291 [MHz]
gbus1_6 3.559683 [MHz]
gbus2_3 2.145764 [MHz]
gbus2_4 2.034764 [MHz]
gbus2_5 1.819238 [MHz]
gbus2_6 1.928233 [MHz]
gbus3_4 3.379720 [MHz]
gbus3_5 1.913611 [MHz]
gbus3_6 1.882454 [MHz]
gbus4_5 3.496662 [MHz]
gbus4_6 2.178134 [MHz]
gbus5_6 2.376696 [MHz]
```

[28] :

	fQ	EC	EJ	alpha	dispersion	\
1	5.369832	353.991815	11.671114	-424.894296	339.569631	
2	5.287303	342.319942	11.671114	-409.153597	261.279371	
3	5.162167	325.060245	11.671114	-386.106665	172.752679	
4	5.093037	315.74898	11.671114	-373.784898	136.236744	
5	5.06007	311.364298	11.671114	-368.009269	121.367243	
6	5.034152	307.942133	11.671114	-363.513283	110.705409	
7	5.018017	305.822908	11.671114	-360.734238	104.495526	
8	5.002207	303.754589	11.671114	-358.025743	98.712255	
9	4.994598	302.761966	11.671114	-356.727215	96.031257	
10	4.987437	301.829611	11.671114	-355.508311	93.567594	
11	4.979585	300.809185	11.671114	-354.175139	90.93076	
12	4.975607	300.293073	11.671114	-353.50119	89.620456	
13	4.972452	299.88403	11.671114	-352.967219	88.593003	
14	4.968176	299.330148	11.671114	-352.244405	87.217134	
15	4.966333	299.091598	11.671114	-351.93318	86.629982	
16	4.96335	298.705685	11.671114	-351.429802	85.686985	

	gbus \
1	[50.20584344455479, 40.82813859764945, -39.959...]
2	[44.53382620893814, 44.14131389822089, -42.836...]
3	[43.11365094770971, 44.198748738291975, -45.10...]
4	[42.621798204162424, 45.529903826869955, -45.0...]
5	[43.17061975602696, 45.61696138237592, -44.863...]
6	[43.05085687766487, 46.66024412626734, -45.706...]
7	[43.15872733820334, 46.69642827423346, -45.807...]
8	[43.244456277018124, 47.34048196557642, -46.25...]
9	[43.47084979034218, 47.5637236254176, -46.5163...]
10	[43.55707740358001, 47.827897678157676, -46.69...]
11	[43.53462238533008, 48.0346705597956, -46.8926...]

```

12 [43.67971952999284, 48.1954621435295, -47.0756...
13 [43.74236679880348, 48.29551324166888, -47.150...
14 [43.726549746882654, 48.405813043744445, -47.2...
15 [43.773805427847016, 48.486449269516555, -47.3...
16 [43.82465139143165, 48.509816005267275, -47.36...

```

		chi_in_MHz	xr MHz	gr MHz
1	[-0.6538815681027647, -9.407394944764015, -5.4...	0.653882	50.205843	
2	[-0.4577014500352182, -7.07773309395745, -4.44...	0.457701	44.533826	
3	[-0.36120866901413295, -4.1952179273222345, -3...	0.361209	43.113651	
4	[-0.3218275523321061, -3.484740917820992, -2.5...	0.321828	42.621798	
5	[-0.31610412498184715, -3.1382201947854016, -2...	0.316104	43.170620	
6	[-0.3038511980250763, -3.0244442947589536, -2...	0.303851	43.050857	
7	[-0.2990165659032797, -2.8818767024429532, -2...	0.299017	43.158727	
8	[-0.2941016018265148, -2.823334726816736, -2.0...	0.294102	43.244456	
9	[-0.2942730197296133, -2.785904700437601, -2.0...	0.294273	43.470850	
10	[-0.2927175921059852, -2.7577498911225646, -2...	0.292718	43.557077	
11	[-0.2894659442874736, -2.7181522866529324, -2...	0.289466	43.534622	
12	[-0.28990796055172546, -2.704767084470196, -2...	0.289908	43.679720	
13	[-0.28956063479068256, -2.6911845381282387, -2...	0.289561	43.742367	
14	[-0.2877627283151042, -2.6701887435709755, -1...	0.287763	43.726550	
15	[-0.2877020425913522, -2.6648646331117862, -1...	0.287702	43.773805	
16	[-0.28726668618817547, -2.6445998906650874, -1...	0.287267	43.824651	

Which gives us the qubits frequency, anharmonicity, and coupling strength to the different connection pads.

We can further check if these parameters converged well. If they have not, we may want to modify our simulation in order to get a more accurate result. We then will want to make modifications to our qubit options, such as `pad_gap` of the qubit, or modifying the size of the connection pads, in order to hit the desired qubit anharmonicity or readout chi values respectively.

```
[29]: c1.plot_convergence();
c1.plot_convergence_chi()
```

```

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:503: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
    self._hfss_variables[variation] = pd.Series(

```

```

Design "Q_Main_q3d" info:
    # eigenmodes      0
    # variations      1

```

```
INFO 09:40PM [hfss_report_full_convergence]: Creating report for variation 0
```

Once the analysis and tuning is complete, we can stop the analysis and the renderer.

```
[30]: c1.sim.close()
```

3.29.2 Eigenmode and EPR

Once each of our qubits have been run through LOM, we can begin to look at the resonant busses and readouts, and larger coupled sections of the chip. One such case could be looking at Q_Main, Q5, and Bus_05. This allows us not only to look at some of the parameters of the individual qubits, but also the bus frequency and if the qubits are coupled (via the bus) to the degree we wish.

3.29.2.0.1 Preparations

We will setup the design and simulation in the same manner as we did previously, but with the methods needed for an eigenmode simulation.

```
[31]: from qiskit_metal.analyses.quantization import EPRanalysis  
eig_qb = EPRanalysis(design, "hfss")
```

(optional) you can tune the wirebond size by directly modifying the renderer options.

```
[32]: eig_qb.sim.renderer.options['wb_size'] = 5
```

Review and update the simulation setup by executing following two cells.

```
[33]: em_p = eig_qb.sim.setup
```

```
[34]: em_p.name = '3Modes'  
em_p.min_freq_ghz = 4  
em_p.n_modes = 3  
em_p.max_passes = 10  
em_p.max_delta_f = 0.1  
em_p.min_converged = 2  
# Design variables can also be added in for direct simulation sweeps.  
em_p.vars = Dict({'Lj1': '13 nH', 'Cj1': '0 fF', 'Lj2': '15 nH', 'Cj2':  
    '0 fF'})  
  
eig_qb.sim.setup
```

```
[34]: {'name': '3Modes',  
       'reuse_selected_design': True,  
       'reuse_setup': True,  
       'min_freq_ghz': 4,
```

```
'n_modes': 3,
'max_delta_f': 0.1,
'max_passes': 10,
'min_passes': 1,
'min_converged': 2,
'pct_refinement': 30,
'basis_order': 1,
'vars': {'Lj1': '13 nH', 'Cj1': '0 fF', 'Lj2': '15 nH', 'Cj2': '0 fF'}}
```

Before we execute the analysis, we want to update the design if needed. For example we may want to modify the junction inductance of the two qubits based on the previous LOM analysis, so they are near the desired frequency. Further, one may want to change the length of the bus after initial simulations to get it to the target frequency:

[35] : `q_main.options.hfss_inductance`

[35] : `'14nH'`

[36] : `Q5.options.hfss_inductance`

[36] : `'14nH'`

[37] : `q_main.options.hfss_inductance = '13nH'`
`Q5.options.hfss_inductance = '15nH'`
`bus_05.options.total_length = '7.5mm'`

`gui.rebuild()`

We can now run the simulation on the specified layout. All unconnected pins are left as shorts, as we are only concerned about simulating the resonant mode of the three components listed.

[38] : `eig_qb.sim.run(name="QMain_Q5_Bus05", components=['Q_Main', ↴ 'Q5', 'Bus_05'], open_terminations=[])`

```
INFO 09:40PM [connect_project]: Connecting to Ansys Desktop API...
INFO 09:40PM [load_ansys_project]:      Opened Ansys App
INFO 09:40PM [load_ansys_project]:      Opened Ansys Desktop v2022.1.0
INFO 09:40PM [load_ansys_project]:      Opened Ansys Project
    Folder:      C:/Users/Bartu/Documents/Ansoft/
    Project:    Project11
INFO 09:40PM [connect_design]:  Opened active design
    Design:    Q_Main_q3d [Solution type: Q3D]
INFO 09:40PM [get_setup]:      Opened setup `Setup` (<class
    'pyEPR.ansys.AnsysQ3DSetup'>)
```

```
INFO 09:40PM [connect]:           Connected to project "Project11" and design
"Q_Main_q3d"
```

```
INFO 09:40PM [connect_design]:  Opened active design
    Design:   QMain_Q5_Bus05_hfss [Solution type: Eigenmode]
WARNING 09:40PM [connect_setup]:      No design setup detected.
WARNING 09:40PM [connect_setup]:      Creating eigenmode default setup.
INFO 09:40PM [get_setup]:          Opened setup `Setup` (<class
'pyEPR.ansys.HfssEMSetup'>)
INFO 09:40PM [get_setup]:          Opened setup `3Modes` (<class
'pyEPR.ansys.HfssEMSetup'>)
INFO 09:40PM [analyze]: Analyzing setup 3Modes
09:43PM 22s INFO [get_f_convergence]: Saved convergences to
C:\Users\Bartu\Desktop\qiskit-metal\tutorials\Appendix A Full design flow
examples\hfss_eig_f_convergence.csv
```

Once the simulation is complete, we can check to see if the convergence was good.

```
[39]: eig_qb.sim.plot_convergences()
```

With the eigenmode simulation complete (and nicely converged) we can run some EPR analysis on the result.

At first we need to define the junctions in terms of name, inductance_variable, capacitance_variable, rectangle that was defined in the rendering to represent the junction port and line that was defined in the rendering to represent the direction of the current through the junction.

```
[40]: eig_qb.del_junction()
eig_qb.add_junction('jj1', 'Lj1', 'Cj1', ↴
    rect='JJ_rect_Lj_Q_Main_rect_jj', line='JJ_Lj_Q_Main_rect_jj_')
eig_qb.add_junction('jj2', 'Lj2', 'Cj2', rect='JJ_rect_Lj_Q5_rect_jj', ↴
    line='JJ_Lj_Q5_rect_jj_')
eig_qb.setup.sweep_variable = 'Lj1'
eig_qb.setup
```

```
[40]: {'junctions': {'jj1': {'Lj_variable': 'Lj1',
    'Cj_variable': 'Cj1',
    'rect': 'JJ_rect_Lj_Q_Main_rect_jj',
    'line': 'JJ_Lj_Q_Main_rect_jj_'},
    'jj2': {'Lj_variable': 'Lj2',
    'Cj_variable': 'Cj2',
    'rect': 'JJ_rect_Lj_Q5_rect_jj',
    'line': 'JJ_Lj_Q5_rect_jj_'},
    'dissipatives': {'dielectrics_bulk': ['main']},
    'cos_trunc': 8,
```

```
'fock_trunc': 7,  
'sweep_variable': 'Lj1'}
```

Note in the previous cell output that the dissipatives have already been defined by default.

Now we can start looking at the EPR values. First we look at the electric field and substrate participation. Then extract the kerr matrix.

```
[41]: eig_qb.run_epr()  
# (pyEPR allows to switch modes: eprd.set_mode(1))
```

```
Design "QMain_Q5_Bus05_hfss" info:  
    # eigenmodes      3  
    # variations      1  
Design "QMain_Q5_Bus05_hfss" info:  
    # eigenmodes      3  
    # variations      1
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-  
packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for  
empty
```

Series will be 'object' instead of 'float64' in a future version. Specify a
dtype explicitly to silence this warning.

```
options=pd.Series(get_instance_vars(self.options)),
```

```
energy_elec_all      = 1.02224728523284e-23  
energy_elec_substrate = 9.30547867824914e-24  
EPR of substrate = 91.0%  
  
energy_mag      = 4.35865303605063e-26  
energy_mag % of energy_elec_all = 0.4%
```

Variation 0 [1/1]

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-  
packages\pyEPR\core_distributed_analysis.py:1096: FutureWarning: The default  
dtype for empty Series will be 'object' instead of 'float64' in a future  
version. Specify a dtype explicitly to silence this warning.
```

```
Ljs = pd.Series({})
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-  
packages\pyEPR\core_distributed_analysis.py:1097: FutureWarning: The default  
dtype for empty Series will be 'object' instead of 'float64' in a future  
version. Specify a dtype explicitly to silence this warning.
```

```
Cjs = pd.Series({})

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.

_0m = pd.Series({})

Mode 0 at 4.45 GHz [1/3]
Calculating _magnetic,_electric

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.

Sj = pd.Series({})

(_E-_H)/_E      _E      _H
99.6%  5.111e-24  2.179e-26

Calculating junction energy participation ration (EPR)
method='line_voltage'. First estimates:
junction      EPR p_0j    sign s_0j    (p_capacitive)
Energy fraction (Lj over Lj&Cj)= 98.01%
jj1            1.7879e-06 (+)      3.636e-08

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.

Qp = pd.Series({})

Energy fraction (Lj over Lj&Cj)= 97.71%
jj2            0.994514 (+)      0.0233366
(U_tot_cap-U_tot_ind)/mean=1.21%
Calculating Qdielectric_main for mode 0 (0/2)

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The
series.append method is deprecated and will be removed from pandas in a
future
version. Use pandas.concat instead.

sol = sol.append(self.get_Qdielectric(
```

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

_Om = pd.Series({})

p_dielectric_main_0 = 0.9102962475590832

Mode 1 at 7.39 GHz [2/3]
Calculating _magnetic,_electric

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

Sj = pd.Series({})

$(_E_H)/_E$	$_E$	$_H$
0.1%	2.384e-24	2.383e-24

Calculating junction energy participation ration (EPR)
method='line_voltage'. First estimates:
junction EPR p_1j sign s_1j (p_capacitive)
Energy fraction (Lj over Lj&Cj)= 94.69%
jj1 0.000135118 (+) 7.57698e-06

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

Qp = pd.Series({})

Energy fraction (Lj over Lj&Cj)=	93.92%	
jj2	0.000502629 (+)	3.25221e-05
(U_tot_cap-U_tot_ind)/mean=0.00%		

Calculating Qdielectric_main for mode 1 (1/2)

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The series.append method is deprecated and will be removed from pandas in a future

version. Use pandas.concat instead.

sol = sol.append(self.get_Qdielectric(

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1240: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

_Om = pd.Series({})

p_dielectric_main_1 = 0.9182273106921761

Mode 2 at 14.81 GHz [3/3]
Calculating _magnetic,_electric

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:976: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

Sj = pd.Series({})

$(_E_H)/_E$	$_E$	$_H$
0.0%	2.095e-24	2.094e-24

Calculating junction energy participation ration (EPR)
method='line_voltage'. First estimates:
junction EPR p_2j sign s_2j (p_capacitive)
Energy fraction (Lj over Lj&Cj)= 81.62%
jj1 1.75751e-05 (+) 3.95788e-06

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:928: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to silence this warning.

Qp = pd.Series({})

Energy fraction (Lj over Lj&Cj)= 79.37%
jj2 6.7549e-05 (+) 1.75522e-05
(U_tot_cap-U_tot_ind)/mean=0.00%

Calculating Qdielectric_main for mode 2 (2/2)

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_distributed_analysis.py:1302: FutureWarning: The series.append method is deprecated and will be removed from pandas in a future

version. Use pandas.concat instead.

sol = sol.append(self.get_Qdielectric(



```
p_dielectric_main_2 = 0.9177931700576385

WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-
packages\pyEPR\project_info.py:226: FutureWarning: The default dtype for u
→empty
Series will be 'object' instead of 'float64' in a future version. Specify a
dtype explicitly to silence this warning.
    options=pd.Series(get_instance_vars(self.options)),

WARNING 09:43PM [ init ]: <p>Error: <class 'IndexError'></p>
```

ANALYSIS DONE. Data saved to:

C:\data-pyEPR\Project11\QMain_Q5_Bus05_hfss\2022-05-14_21-43-23.npz

Differences in variations:

Variation 0

Starting the diagonalization
Finished the diagonalization

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-  
packages\pyEPR\core_quantum_analysis.py:715: FutureWarning: Support for  
    ↪multi-  
dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be  
    ↪removed in  
a future version. Convert to a numpy array before indexing instead.  
    result['Q_coupling'] = self.Qm_coupling[varyation][self.  
    ↪Qm_coupling[varyation]  
.columns[junctions]] [modes]#TODO change the columns to junctions
```

```
WARNING:py.warnings:C:\Users\Bartu\anaconda3\envs\metal\lib\site-packages\pyEPR\core_quantum_analysis.py:720: FutureWarning: Support for multi-dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be removed in a future version. Convert to a numpy array before indexing instead.
```

```

result['Qs'] =
self.Qs[vary][self.PM[vary].columns[junctions]][modes] #TODO
    ↵change
the columns to junctions

Pm_norm=
modes
0      1.024641
1      1.032057
2      1.127610
dtype: float64

Pm_norm idx =
      jj1      jj2
0  False   True
1  False  False
2  False  False
*** P (participation matrix, not normlzd.)
      jj1      jj2
0  0.000002  0.971834
1  0.000135  0.000503
2  0.000018  0.000068

*** S (sign-bit matrix)
      s_jj1  s_jj2
0      1      1
1      1      1
2      1      1
*** P (participation matrix, normalized.)
      1.7e-06      1
      0.00014    0.0005
      1.8e-05  6.8e-05

*** Chi matrix 01 PT (MHz)
Diag is anharmonicity, off diag is full cross-Kerr.
      225      0.378      0.102
      0.378  0.000168  9.04e-05
      0.102  9.04e-05  1.22e-05

*** Chi matrix ND (MHz)
      252      0.342      0.0934
      0.342  0.000137  9.1e-05
      0.0934  9.1e-05  1.21e-05

```

```
*** Frequencies 01 PT (MHz)
```

```
0      4225.570083
1      7391.181069
2      14812.013576
dtype: float64
```

```
*** Frequencies ND (MHz)
```

```
0      4212.629439
1      7391.184208
2      14812.016368
dtype: float64
```

```
*** Q_coupling
```

```
Empty DataFrame
Columns: []
Index: [0, 1, 2]
```

3.29.2.0.2 Mode frequencies (MHz)

Numerical diagonalization

```
Lj1      13
0      4212.63
1      7391.18
2      14812.02
```

3.29.2.0.3 Kerr Non-linear coefficient table (MHz)

Numerical diagonalization

Lj1	0	1	2
13	0 252.33 3.42e-01 9.34e-02	1 0.34 1.37e-04 9.10e-05	2 0.09 9.10e-05 1.21e-05

From the analysis results we can determine the qubits anharmonicities and coupling strength. Once the analysis and tuning is complete, we can close the connection to Ansys.

```
[42]: eig_qb.sim.close()
```

3.29.2.1 Rendering to a GDS File

Once all of the tuning is complete, we will want to prepare a GDS file so we can create a mask and fabricate our chip. We first create a gds render instance.

```
[43]: full_chip_gds = design.renderers.gds
```

The various options for the gds renderer can also be checked and changed as necessary. A key option is the gds file which holds the cells for your junction ebeam design. Make sure this is pointing at the correct file so they are placed in your final mask at the appropriate locations.

[44]: full_chip_gds.options

```
[44]: {'short_segments_to_not_fillet': 'True',
       'check_short_segments_by_scaling_fillet': '2.0',
       'gds_unit': 0.001,
       'ground_plane': 'True',
       'negative_mask': {'main': []},
       'fabricate': 'False',
       'corners': 'circular bend',
       'tolerance': '0.00001',
       'precision': '0.000000001',
       'width_LineString': '10um',
       'path_filename': '../resources/Fake_Junctions.GDS',
       'junction_pad_overlap': '5um',
       'max_points': '199',
       'cheese': {'datatype': '100',
                  'shape': '0',
                  'cheese_0_x': '25um',
                  'cheese_0_y': '25um',
                  'cheese_1_radius': '100um',
                  'view_in_file': {'main': {1: True}},
                  'delta_x': '100um',
                  'delta_y': '100um',
                  'edge_nocheese': '200um'},
       'no_cheese': {'datatype': '99',
                     'buffer': '25um',
                     'cap_style': '2',
                     'join_style': '2',
                     'view_in_file': {'main': {1: True}}},
       'bounding_box_scale_x': '1.2',
       'bounding_box_scale_y': '1.2'}
```

[45]: full_chip_gds.options['path_filename'] = '../resources/Fake_Junctions.GDS'
full_chip_gds.options['no_cheese']['buffer'] = '50um'

[46]: full_chip_gds.export_to_gds('Full_Chip_01.gds')

Bibliography

- [1] Qiskit textbook, ibm. <https://qiskit.org/textbook/ch-quantum-hardware/cQED-JC-SW.html>.
- [2] Qiskit metal documentation, ibm. <https://qiskit.org/documentation/metal/>.
- [3] S. M. Girvin A. Wallraff A. Blais, A. L. Grimsmo. Circuit quantum electrodynamics. 2020.
- [4] J. M. Raimond S. Haroche, M. Brune. From cavity to circuit quantum electrodynamics. 2020.
- [5] M. Takita A. Corcoles J. Gambetta Z. Minev, T. McConkey. Circuit quantum electrodynamics (cqed) with modular quasi-lumped models. 2021.
- [6] S. O. Mudhada P. Reinhold A. Diringer M. H. Devoret Z. Minev, Z. Leghtas. pyepr: The energy-participation-ratio (epr) open-source framework for quantum device design. 2021.