scqubits: a Python package for superconducting qubits

Peter Groszkowski[1] and Jens Koch[2]

[1]Pritzker School for Molecular Engineering, University of Chicago, 5640 South Ellis Avenue, Chicago, IL 60637, USA
[2]Department of Physics and Astronomy, Northwestern University, Evanston, IL 60208, USA

scqubits is an open-source Python package for simulating and analyzing superconducting circuits. It provides convenient routines to obtain energy spectra of common superconducting qubits, such as the transmon, fluxonium, flux, cos(2ϕ) and the 0-π qubit. scqubits also features a number of options for visualizing the computed spectral data, including plots of energy levels as a function of external parameters, display of matrix elements of various operators as well as means to easily plot qubit wavefunctions. Many of these tools are not limited to single qubits, but extend to composite Hilbert spaces consisting of coupled superconducting qubits and harmonic (or weakly anharmonic) modes. The library provides an extensive suite of methods for estimating qubit coherence times due to a variety of commonly considered noise channels. While all functionality of scqubits can be accessed programatically, the package also implements GUI-like widgets that, with a few clicks can help users both create relevant Python objects, as well as explore their properties through various plots. When applicable, the library harnesses the computing power of multiple cores via multiprocessing. scqubits further exposes a direct interface to the Quantum Toolbox in Python (QuTiP) package, allowing the user to efficiently leverage QuTiP's proven capabilities for simulating time evolution.

Jens Koch: jens-koch@northwestern.edu

1 Introduction

Superconducting qubits [7, 10, 17, 27] have secured the rank of one of the most promising and widely researched hardware architectures for quantum information processing. All devices in this category are relatively simple circuits, and display genuine quantum properties such as discrete energy spectra and quantum-coherent time evolution. Coupling superconducting qubits to external electromagnetic fields enables the realization of gate operations as well as quantum-state readout using the framework of circuit quantum electrodynamics (circuit QED) [3, 4, 26].

The computation of energy spectra, eigenstates, and matrix elements of relevant operators is a key prerequisite for the design and fabrication of superconducting qubits, as well as for the quantitative analysis of experimental data collected in state-of-the-art experiments. Circuit quantization [6, 9, 25] provides the systematic framework for deriving the Hamiltonian operator which describes a given circuit mathematically. However, with the exception of the simple LC oscillator and the limiting behavior of nonlinear circuits in specific parameter regimes, computation of qubit spectra cannot be accomplished analytically, but rather requires numerical solution of Hermitian eigenvalue problems.

The scqubits package provides a user-friendly, object-oriented Python library of the most common superconducting qubits. It facilitates automatic construction of circuit Hamiltonians in an appropriate basis, provides high-level routines for finding eigenenergies, eigenstates, and matrix elements, and allows the user to quickly visualize these quantities as a function of external parameters. While the scqubits routines for numerics and plotting rely heavily on NumPy, SciPy, and Matplotlib, no detailed knowledge of these internals is required from

the user to employ scqubits efficiently. In this way, the package helps lower initial barriers encountered by new researchers entering the field, and can be easily integrated for educational purposes. At the same time, the library aims to fulfill the role of a unifying tool for expert workers in the field, enabling quick and efficient investigations of a wide variety of circuit-QED systems, and the interactive exploration of their behavior in different parameter regimes.

The analysis of interesting circuit-QED systems invariably involves the consideration of coupled systems, composed of qubits on one hand, and harmonic modes on the other hand (realized, for example, as on-chip transmission-line resonators or 3d cavities). The scqubits library simplifies the setup of such composite Hilbert spaces and grants a seamless interface to the well-established QuTiP [14, 15] framework which can be leveraged for the simulation of time evolution.

This paper is organized as follows. In the next section, we provide an overview of the library and its core functionality. In Sec. 3 we discuss how to construct and analyze composite Hilbert space systems that may consist of multiple qubits and/or resonators. Next, in Sec. 4 we present how scqubits lets users perform parameter sweeps and easily explore how properties of a composite system vary as circuit parameters or control fields change. In Sec. 5 we review how scqubits can be used to estimate coherence times of different qubits, and how those coherence times can be visualized. After that, in Sec. 6 we provide a brief overview of the interactive exploration capabilities of the library, that allow users to study properties of various systems in ways that require very little actual programming. Finally, we summarize and conclude in Sec. 8.

2 Overview of the scqubits library

This section gives a broad overview and introduces the main building blocks of scqubits based on concrete examples of their usage. We stress that a jupyter notebook containing all of the source code in this manuscript can be found in the github examples repository (see Sec. 7). As a regular Python package, scqubits is imported via

```
import scqubits as scq
```

The simplest way to explore spectral properties of individual superconducting qubits is to invoke the dedicated graphical user interface via

```
scq.GUI()
```

which outputs the display shown in Fig. 1. With the exception of the initial call, usage of this interface requires no further knowledge of Python and is particularly well-suited for beginners. In the following we describe the more powerful and flexible programmatic usage of scqubits.

As an illustration of basic scqubits functionality, we consider the transmon qubit. Like all qubit types implemented in scqubits, `Transmon` and its flux-tunable variant `TunableTransmon` are realized as Python classes. Each class instance stores all relevant circuit parameters and control-field values, and provides a collection of methods used for common computations and visualization. (See Table 3 for a summary of the most commonly used qubit class methods.) An instance of the `TunableTransmon` class is created by the following call which provides all necessary system parameters for initialization:

```
tmon = scq.TunableTransmon(
    EJmax=30.0,
    EC=1.2,
    d=0.01,
    flux=0.0,
    ng=0.0,
    ncut=30
)
```

The initialization arguments include the relevant circuit parameters: for a flux-tunable transmon, these are the maximum Josephson energy $E_{J\max}$ from the SQUID loop, charging energy E_C, and offset charge n_g (further details are provided in appendix A.III). If scqubits is used inside a jupyter notebook, then

```
transmon = scq.TunableTransmon.create()
```

is an alternative way to create and initialize a `TunableTransmon` instance. The resulting graphical interface offers simple widgets to enter required parameters and displays the transmon circuit for reference. By default, energies are assumed to be

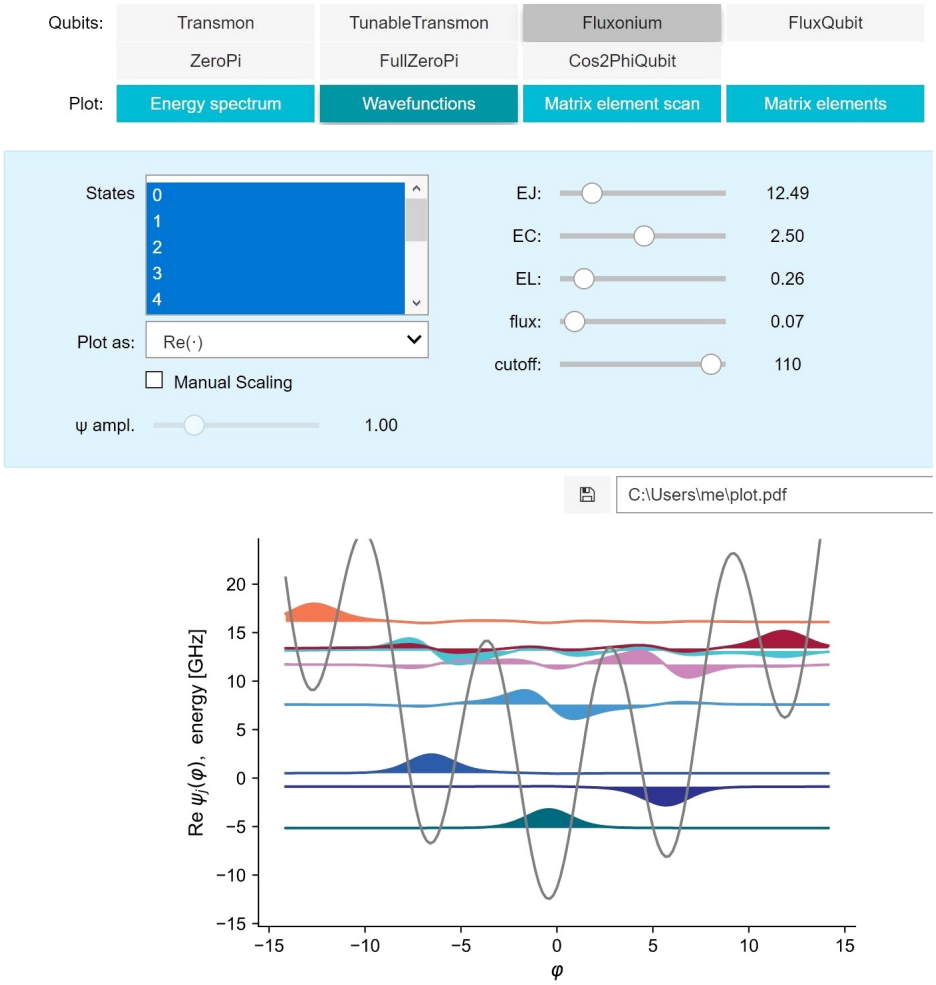


Figure 1: scqubits graphical user interface for exploring properties of individual superconducting qubits.

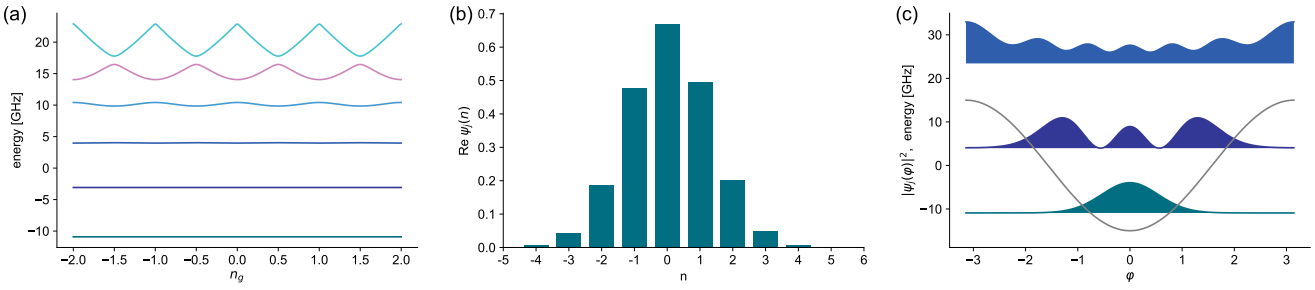


Figure 2: Visualization of transmon eigenenergies and eigenstates. (a) Plot of the lowest six transmon energy eigenvalues as a function of the offset charge n_g. (b) Plot of wavefunction amplitudes in the discrete charge basis, using `plot_n_wavefunction()`. The slight asymmetry in the amplitudes with respect to $n = 0$ originates from the choice of a nonzero offset charge n_g. (c) `plot_phi_wavefunction()` graphs wavefunctions in the φ-basis, along with the underlying potential energy. Wavefunctions are offset vertically according to their eigenenergies.

given as frequencies in units of GHz, although this can be easily modified – see Sec. 2.4.

Finding eigenenergies and eigenstates of superconducting qubits invariably involves truncation of the infinite-dimensional Hilbert space. In each qubit class, this can be controlled by setting the appropriate truncation parameter, such as `ncut` for the transmon qubit. While typical values are suggested in each widget, convergence with respect to this cutoff (and similar cutoffs in other qubit classes) must be established by the user (see 2.1.1 below for a more detailed discussion).

2.1 Computing and plotting energy spectra

The energy eigenvalues of the transmon Hamiltonian are obtained by calling the `eigenvals` method. The optional parameter `evals_count` specifies the desired number of eigenenergies:

```
tmon.eigenvals(evals_count=12)
```

Execution of this line yields a NumPy array of the lowest twelve energy eigenvalues. To plot eigenenergies as a function of one of the qubit parameters (here, `EJmax`, `EC`, `flux`, or `ng`), we generate an array of parameter values of interest, and then call the method `plot_evals_vs_paramvals`. The latter takes as arguments the name of the parameter to be varied, an array of parameter values, and optionally the eigenvalue number. The following is an example for plotting the lowest six eigenenergies as a function of offset charge `ng` for 220 equally spaced points in the interval $n_g \in [-2, 2]$:

```
ng_list = np.linspace(-2, 2, 220)
tmon.plot_evals_vs_paramvals('ng', ng_list,
                             evals_count=6)
```

Figure 2(a) shows the resulting output. Plotting routines in scqubits rely on matplotlib, and generally return a tuple of a `Figure` and an `Axes` object to enable post-processing of the plot, if desired.

The full eigensystem consisting of both eigenvalues and eigenvectors is obtained through the method `eigensys`:

```
evals, evecs = tmon.eigensys()
```

For the transmon qubit, this calculation is based on[1] `scipy.linalg.eigh` which performs diagonalization of the Hamiltonian matrix expressed in the charge basis. As dictated by the SciPy package, the eigenvector corresponding to the `j`-th eigenvalue is the Numpy array `evecs.T[j]`.

2.1.1 Hilbert space truncation and convergence

As mentioned above, obtaining qubit spectra requires truncating the Hilbert space to some finite dimension. Users can control the Hilbert space dimension used by scqubits, by passing appropriate values for `ncut` and/or `grid` parameters to qubit class constructors [2]. These parameters effectively define the number of basis states that are used during diagonalization. In qubits with multiple degrees of freedom, users need to set a cutoff independently for each one.

Choosing cutoffs or grid sizes that are too small may lead to inaccurate results. The specific parameter values required for convergence naturally depend on the qubit type, circuit energies, as well as how many eigenenergies and/or eigenvectors the user wishes to obtain. While in some simple cases (e.g., transmon qubit - see appendix A.III) one can establish stringent cutoff requirements, for most circuits convergence must be established by trial and error.

A heuristic approach to this end is to repeat calculations with successive increases in cutoffs and grid sizes until results are essentially unchanged (within the desired accuracy). Once this is achieved, errors are typically limited by the default tolerances set by the `scipy` routines which scqubits internally uses for matrix diagonalization.

2.2 Plotting wavefunctions

The transmon qubit is a circuit with a single degree of freedom. Hence, its wavefunctions are readily plotted in the two bases natural for the transmon: the discrete charge basis $\psi_j(n) = \langle n|\psi_j\rangle$, and

[1] In some cases (e.g., $0-\pi$ or the $\cos 2\phi$ qubit), scqubits, uses `scipy.sparse.linalg.eigsh` for diagonalization.

[2] In some qubits with multiple degrees of freedom, these variable names may be slightly modified to reflect which degrees of freedom are being addressed. See the API documentation [1] as well as the Appendices for details.

the continuous phase basis $\psi_j(\varphi) = \langle\varphi|\psi_j\rangle$. The `TunableTransmon` class offers two methods for this purpose:

```
tmon.plot_n_wavefunction(which=0, mode='real');
tmon.plot_phi_wavefunction(which=[0,2,6],
                           mode='abs_sqr');
```

Figure 2(b-c) shows the generated graphs. Above, the keyword argument `which` specifies the selection of wavefunctions to plot. In the first case, `which =0` indicates that only the ground state wavefunction (index 0) should be displayed. In the second case, `which` is assigned a list consisting of multiple such indices, resulting in a plot showing several wavefunctions at once. The `mode` option determines how to plot the wavefunction which is generally complex-valued. The self-explanatory options are `'real'`, `'imag'`, `'abs'`, and `'abs_sqr'`.

For one-dimensional wavefunctions in φ basis ("position" basis), the plot mimics the textbook format widely used for 1d spectra: wavefunctions of the eigenstates are shown alongside the potential-energy function, and are offset vertically by their corresponding eigenenergies.

2.3 Evaluating and visualizing matrix elements

Matrix elements of qubit operators play an important role in determining coupling strengths between a qubit and another quantum system. They are also critical for the sensitivity of the qubit to various sources of noise. In the case of the transmon qubit, for example, matrix elements of the charge operator are of frequent interest. This charge operator, $\hat{n}$, is accessible through the class method `n_operator`. Evaluation of the matrix elements for the current set of parameters is implemented by the method `matrixelement_table`. An overview plot of matrix elements with respect to the lowest ten transmon eigenstates is obtained by

```
tmon.plot_matrixelements('n_operator',
                         evals_count=10,
                         show_numbers=True)
```

where the reference to the operator is provided in string format. The resulting plot is shown in Fig. 3.

Occasionally, it is further useful to plot matrix elements as a function of an external parameter.

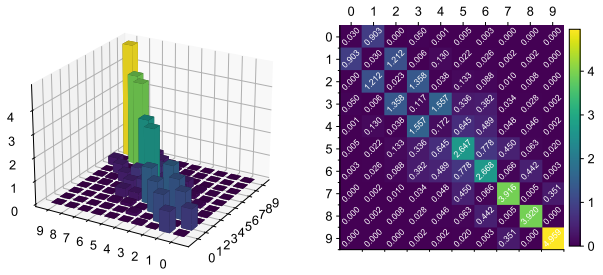


Figure 3: Visualization of the matrix elements of the transmon charge operator $\langle\psi_i|\hat{n}|\psi_j\rangle$, evaluated with respect to the transmon eigenstates using `plot_matrixelements`. Numerical values can be included with the option `show_numbers=True`.

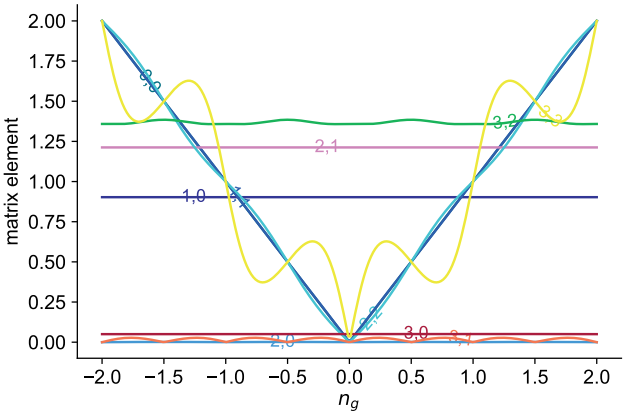


Figure 4: Plot of select matrix elements as a function of the external offset-charge parameter n_g.

This is accomplished by `plot_matelem_vs_paramvals` Applied to the transmon charge matrix elements, we can easily visualize the dependence on the offset charge n_g:

```
ng_list = np.linspace(-2, 2, 220)
tmon.plot_matelem_vs_paramvals('n_operator',
                               'ng', ng_list,
                               select_elems=4)
```

Here, the names of both the operator and the external parameter are given as string arguments, followed by the array of values for the external parameter. Finally, `select_elems=4` specifies that all matrix elements $\langle\psi_i|n|\psi_j\rangle$ with $0 \le i, j \le 3$ are requested in the plot. The output is shown in Fig. 4.

2.4 Units

scqubits allows the user to associate specific units with the energy values they provide when instantiating various qubit classes. The knowledge by scqubits of implied units is necessary for calculations that involve coherence time estimations (see Sec. 5), but also come in handy for automatic labeling of plot axes. All energies are assumed to be expressed in terms of frequencies (not angular frequencies), with the default being GHz. Other supported units are MHz, kHz and Hz. A list containing these possible choices can be shown with the `show_supported_units` function. The currently set units can be obtained with the `get_units` function, and changed with `set_units`, like so:

```
scq.get_units()
scq.set_units('MHz')
```

scqubits also includes several helper functions for convenient conversion between the currently set units and Hz, which include `to_standard_units`, `from_standard_units` and `units_scale_factor`.

3 Composite Hilbert spaces and interface with QuTiP

An important aspect of modeling superconducting circuits is the ability to study composite systems. scqubits provides an easy mechanism to explore setups that may consist of multiple qubits as well as harmonic (and weakly anharmonic) oscillators. Along with providing an easy way of constructing composite-system Hilbert spaces, and calculating and visualizing their many properties, scqubits also allows for easy exporting of effective Hamiltonians to QuTiP, an established toolbox for studying stationary and dynamical properties of closed and open quantum systems. At the heart of this functionality is the `HilbertSpace` class, which provides the data structures and methods for handling composite Hilbert space objects, and which we briefly explore in the sections below.

3.1 Example system: two transmons coupled to a harmonic mode

Transmon qubits can be capacitively coupled to a common harmonic mode, realized by an LC oscillator or a transmission-line resonator. The Hamiltonian describing such a composite system is given by

$$\hat{H} = E_{\rm osc}\hat{a}^\dagger\hat{a} + \sum_{j=1,2}\hat{H}_{\rm tmon,j} + \sum_{j=1,2} g_j\hat{n}_j(\hat{a}+\hat{a}^\dagger), \tag{1}$$

where $j = 1, 2$ enumerates the two transmon qubits, $E_{\rm osc}$ is the single-photon energy for the resonator. Furthermore, $\hat{n}_j$ is the charge number operator for qubit j, and g_j is the coupling strength between qubit j and the resonator.

The first step consists of creating the objects describing the individual building blocks of the full Hilbert space. Here, these will be the two transmons and one oscillator:

```
tmon1 = scq.Transmon(
    EJ=40.0,
    EC=0.2,
    ng=0.3,
    ncut=40,
    truncated_dim=4
)

tmon2 = scq.Transmon(
    EJ=15.0,
    EC=0.15,
    ng=0.0,
    ncut=30,
    truncated_dim=4
)

resonator = scq.Oscillator(
    E_osc=4.5,
    truncated_dim=4
)
```

The significance of `truncated_dim` lies in a simple hierarchical diagonalization scheme. Specifically, each subsystem is diagonalized separately in step one. Subsequently, the lowest few bare subsystem eigenstates up to truncation level `truncated_dim` are fed forward into `HilbertSpace`.

3.2 Creating the `HilbertSpace` object

The desired `HilbertSpace` object can be created in two ways: either by utilizing the GUI displayed via `hs = scq.HilbertSpace.create()`, or programmatically by initializing the `HilbertSpace` object with a list of all subsystems and then specifying individual interaction terms:

```
hs = scq.HilbertSpace([tmon1,tmon2,resonator])
```

3.3 Specifying interactions

Interaction terms describing the coupling between subsystems (or modifying a single subsystem itself) can be specified via the method `add_interaction` in three different ways.

1. Operator-product based interface: Interaction terms involving multiple subsystems $S = 1, 2, 3, \ldots$ are often of the form

$$\hat{V} = g\,\hat{A}_1\hat{A}_2\hat{A}_3\cdots \qquad \text{or}$$
$$\hat{V} = g\,\hat{B}_1\hat{B}_2\hat{B}_3\cdots + g^*(\hat{B}_1\hat{B}_2\hat{B}_3\cdots)^\dagger$$

where the operators $\hat{A}_j$, $\hat{B}_j$ act on subsystem j. (In the first case, the operators $\hat{A}_j$ are expected to be hermitian.) This structure is captured in the following way:

```
# coupling resonator-tmon1
g1 = 0.1
operator1 = tmon1.n_operator()
operator2 = resonator.creation_operator() +
            resonator.annihilation_operator()

hs.add_interaction(
    g=g1,
    op1=(operator1, tmon1),  # (matrix, subsys)
    op2=(operator2, resonator)
)

# coupling resonator-tmon2
g2 = 0.2
hs.add_interaction(
    g=g2,
    op1=tmon2.n_operator,    # class method
    op2=resonator.creation_operator,
    add_hc=True
)
```

In this operator-product interface, `op1`, `op2`,... are either of the form `(<array>, <subsystem>)`, i.e., a tuple with an array or sparse matrix in the first position, and the corresponding subsystem in the second position; or of the form `<callable>`, i.e., the operator is provided as a callable method which will automatically yield the subsystem the operator function is bound to. (These two choices can be mixed and matched.) The option `add_hc=True` adds the hermitian conjugate to the specified interaction term.

Note that interactions based on only one operator are possible (simply drop all but the `op1` entry). One example use case of this is the creation of a higher-order non-linearity $a^\dagger a^\dagger a^\dagger aaa$ in a Kerr oscillator.

2. String-based interface: The `add_interaction` method can also be used to define the interaction in string form, by providing an expression that can be evaluated by the Python interpreter.

```
hs = scq.HilbertSpace([tmon1, tmon2, resonator])
g3 = 0.1

hs.add_interaction(
    expr="g3 * cos(n) * adag",
    op1=("n", tmon1.n_operator(), tmon1),
    op2=("adag", resonator.creation_operator),
    add_hc=True
)
```

Here, `expr` is a string used to define the interaction as a Python expression. It may use variables that are already defined globally, and operators given by the names provided in `op1`, `op2`, ...

3. `Qobj` interface: Finally, `add_interaction` can be used to directly add a QuTiP `Qobj` that has already been properly wrapped with identities:

```
import qutip as qt

# Generate a Qobj
g = 0.1
a = qt.destroy(4)
kerr = a.dag() * a.dag() * a * a
id = qt.qeye(4)
V = g * qt.tensor(id, id, kerr)

```

```
hs.add_interaction(qobj=V)
```

3.4 Obtaining the Hamiltonian and spectrum

With the interactions specified, the full Hamiltonian of the coupled system can be obtained via the method `hamiltonian`,

```
dressed_hamiltonian = hs.hamiltonian()
```

which represents H in the basis of the bare product states composed of subsystem eigenstates. Since the Hamiltonian obtained this way is a proper `Qobj`, it can easily be handed over to QuTiP's time evolution routines such as `mesolve`. Eigenenergies and eigenstates can now either be obtained via the usual scqubits methods,

```
evals = hs.eigenvals()  # or
evals, evecs = hs.eigensys()
```

or by invoking QuTiP methods on the Hamiltonian itself, e.g., `hs.hamiltonian.eigenstates()`.

4 Sweeping over external parameters

Determining the dependence of physical observables on one or multiple external parameter(s) is a common way to gain intuition for the properties and behavior of a system. Such parameter sweeps can be performed with scqubits on multiple levels: (1) at the level of a single qubit, and (2) at the level of a composite quantum system.

At the single-qubit level, each qubit class provides several methods that enable producing parameter sweep data and plots. Central quantities of interest, in this case, are energy eigenvalues and matrix elements – in particular, their dependence on parameters like flux or offset charge. The relevant methods available for every implemented qubit class are:

- `get_spectrum_vs_paramvals`
  sweep eigenvalues and eigenvectors
- `get_matelements_vs_paramvals`
  sweep matrix elements
- `plot_evals_vs_paramvals`
  plot eigenenergy sweep
- `plot_matelem_vs_paramvals`
  plot matrix element sweep

4.1 Creating a `ParameterSweep` object

Composite Hilbert spaces, as implemented by `HilbertSpace` objects, are naturally richer than individual qubits. A variety of parameter sweeps can be considered, including multi-dimensional sweeps over a collection of different parameters. For flexible parameter scans, scqubits provides the `ParameterSweep` class. To illustrate its usage, we reuse a composite Hilbert space akin to the one presented above: two tunable transmon qubits capacitively coupled to an oscillator.

The `ParameterSweep` class facilitates computation of spectra as function of one or multiple external parameter(s). For efficiency in computing a variety of derived quantities and creating plots, the computed bare and dressed spectral data are stored internally. A `ParameterSweep` object is initialized by providing the following parameters:

1. `hilbertspace`: a `HilbertSpace` object that describes the quantum system of interest
2. `paramvals_by_name`: a dictionary that maps each parameter name (string) to an array of parameter values
3. `update_hilbertspace`: a function that defines how parameter changes affect the system
4. `subsys_update_info`: (optional) for potential speed-up, specify which subsystems undergo changes as each of the parameters is varied
5. `deepcopy`: (optional) determines whether the HilbertSpace object and all constituents should be duplicated and disconnected from the global objects
6. `num_cpus`: (optional) number of CPU cores requested for the sweep evaluation

These ingredients all enter as initialization arguments of the `ParameterSweep` object. Once initialized, spectral data is generated and stored.

In our example, we consider the strength of a global magnetic field as the parameter to be changed. This field determines the magnetic fluxes for both qubits, in proportions according to their SQUID loop areas. We will reference the flux for transmon 1, and express the flux for transmon 2 in terms of it via an area ratio. In addition, we will vary the offset charge of transmon 2.

The following code illustrates this functionality:

```
# combine parameter names and values
```

```
# in a dictionary
paramvals_by_name = {
    "flux": np.linspace(0.0, 2.0, 171),
    "ng": np.linspace(-0.5, 0.5, 49)
}

area_ratio = 1.2
def update_hilbertspace(flux, ng):
    # function that defines how Hilbert space
    # components are updated
    tmon1.flux = flux
    tmon2.flux = area_ratio * flux
    tmon2.ng = ng

# dictionary specifying which subsystems are
# affected by changing each parameters
subsys_update_info = {"flux": [tmon1, tmon2],
                      "ng": [tmon2]}

# create the ParameterSweep object
sweep = scq.ParameterSweep(
    hilbertspace=hs,
    paramvals_by_name=paramvals_by_name,
    update_hilbertspace=update_hilbertspace,
    evals_count=20,
    subsys_update_info=subsys_update_info,
    num_cpus=4
)
```

In the code above, `update_hilbertspace` directly manipulates transmon instances via their global references. Alternatively, `HilbertSpace` constituents can be accessed via `sweep.hilbertspace[<id_str>]` where `id_str` is a string identifier either provided explicitly at initialization of object instances, or autogenerated by scqubits. (Details and examples of this functionality are available in the documentation and example notebooks – see Sec. 7.)

4.2 Generated `ParameterSweep` data

Much of the computed data that is stored and immediately retrievable after this sweep. The stored data is accessed as if `ParameterSweep` were a Python `dict` with the following string keys:

1. `"evals"`, `"evecs"`:
   dressed eigenenergies and eigenstates
2. `"bare_evals"`, `"bare_evecs"`:
   bare eigenenergies and eigenstates for each subsystem
3. `"lamb"`, `"chi"`, `"kerr"`:
   dispersive energy coefficients

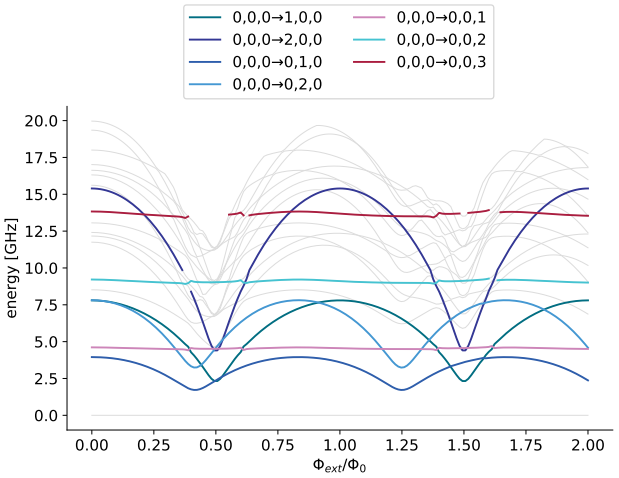
Data are returned as a `NamedSlotsNdarray`, a subclass of the regular NumPy `ndarray`. It features several convenient new slicing options, such as slicing by parameter name, or slicing by reference to a particular parameter value. See [2] for a more more detailed discussion as well as examples.

4.3 Transition plots

Energy spectra obtained in single-tone or two-tone spectroscopy always represent transition energies, rather than absolute energies of individual eigenstates. To generate data mimicking this, appropriate differences between eigenenergies must be taken.

The methods for generating transition energy data and plotting them are `transitions` and `plot_transitions`. To create a plot, we first "preslice" the `ParameterSweep` instance which specifies a sweep along a single axis. Then, the `plot_transitions` method can be called[3]:

```
sweep["ng":0.0].plot_transitions();
```



When coloring transitions according to the dispersive-limit product state labels, one potential artifact is nearly unavoidable: whenever states undergo avoided crossings and the dispersive limit breaks down, coloring must discontinuously switch from one branch to another. scqubits attempts to interrupt coloring in such regions. However, if the avoided crossing occurs over a range comparable to

[3] A notebook with exact parameters used to generate this plot is included in the scqubits-examples github repository – see Sec. 7.

the parameter value spacing, then discontinuities from connecting separate branches will remain visible.

Transition plot options: The generated transition plot above is based on a number of default settings, including: (i) the origin of each transition is the system's ground state, (ii) single-photon transitions are plotted in light grey and (iii) transitions within each individual subsystem are marked separately in color and accompanied by a legend. This is possible in regions where the dispersive approximation holds, i.e., hybridization between subsystems remains weak. Labels in the legend are excitation levels of individual systems: ((0,0,0), (1,0,0)) denotes a transition from the ground state to the state with subsystem 1 in the first excited state, and subsystems 2 and 3 in their respective ground states.

Many aspects of transition energy plots can be changed. The following illustrates a subset of options that change which transitions are plotted and how.

`coloring`: Coloring based on dispersive-transition identification can be switched off by setting `coloring="plain"`.

`subsystems`: By default, dispersive-transition coloring includes all subsystems. If only transitions for a single or smaller set of subsystem(s) should be highlighted, then these can be specified in list form, e.g. `subsystems=[tmon1]`.

`initial,final`: The ground state is the default origin for all transitions. In case of thermal excitations, other states can be of interest as initial states. Specification of an alternative initial state uses dispersive labeling of states, e.g., `initial=(1,0,0)` uses the 1st excited of the first subsystem as the initial state.

`photon_number`: To model n-photon transitions, setting `photon_number=`n yields transition energies divided by n.

`sidebands`: For `sidebands=True` sideband transitions with multiple subsystems changing excitation levels are included in color highlighting and legend.

4.4 Custom sweep data

`ParameterSweep` automatically generates data commonly needed in studying a multi-component quantum system. Other quantities of interest can be generated by defining a custom sweep function,

```
def custom_func(paramsweep, paramindex_tuple,
                paramvals_tuple, **kwargs):
    ...
    return data
```

This function returns the data to be calculated for each parameter choice. The custom sweep is then performed upon calling

```
<ParameterSweep>.add_sweep(custom_func,
                           "custom name")
```

The computed data is subsequently accessible via

```
<ParameterSweep>["custom name"]
```

5 Estimation of coherence times

scqubits provides extensive functionality that allows users to estimate coherence times of various qubits. Very general methods `t1` and `tphi` are implemented for each noisy qubit, which can be used for calculations of depolarization as well as pure dephasing times for almost arbitrary noise processes (see sections 5.1 and 5.2 for more details). Furthermore, a large variety of predefined methods corresponding to more specific noise channels (e.g., dephasing due to $1/f$ charge noise, or depolarization from coupling to a transmission line) are implemented as well (for a list, see Table 1).

Due to different qubit properties and circuit topologies, each qubit is affected by some specific subset of these predefined noisy processes. To see which channels are currently implemented for any given qubit, one can run the command (here shown for the case of a transmon)

```
tmon.supported_noise_channels()
```

```
['tphi_1_over_f_flux',
 'tphi_1_over_f_cc',
 'tphi_1_over_f_ng',
 't1_capacitive',
 't1_flux_bias_line',
 't1_charge_impedance']
```

This returned list contains methods with self-explanatory names that either start with `t1` or `tphi` and represent the depolarization or pure dephasing processes, respectively. Each of these methods can then be called directly to obtain the corresponding coherence time (which will take into account the current units setting – see section 2.4). For example, in order to calculate the pure dephasing time due to $1/f$ flux noise, one can simply execute

```
tmon.tphi_1_over_f_flux()
```

Each available method can be provided with a set of custom parameters that give the user a means to fine-tune various noise process properties (e.g., bath temperature, $1/f$ flux-noise strength, etc.). There are also common method options that specify which qubit levels should be considered (levels 0 and 1 are assumed by default), whether a rate instead of a time should be returned, or whether depolarization rates should include both upwards and downwards transitions. A more complicated method call could therefore take the form

```
tmon.t1_charge_impedance(i=3, j=1,
                         Z=50,
                         T=0.100,
                         get_rate=True,
                         total=False)
```

This returns a depolarization rate (not time) between levels 3 and 1, due to the qubit coupling to a $50\,\Omega$ transmission line, at a temperature of 100 mK. In this case, the impedance (parameter `Z`) can be a constant, or alternatively an angular frequency-dependent Python function that will be evaluated at the frequency difference between levels i and j.

In the next sections we briefly outline the basic physics and assumptions underlying the estimation of the various pure dephasing and depolarization times.

5.1 Dephasing due to $1/f$ noise

Dephasing noise leads to loss of coherence, i.e., the relative phases relevant in superpositions of multiple states are lost over time. One of the most important kinds of noise affecting superconducting qubits is $1/f$ noise, which leads to slow fluctuations of the energy-level spacing. The spectral density function characterizing this noise is given by

$$S(\omega) = \frac{2\pi A_\lambda^2}{|\omega|}. \tag{2}$$

Here, A_λ corresponds to the amplitude or strength of the particular noise channel λ, such as charge or flux. scqubits uses sensible default values for this quantity based on the literature. Alternative values, however, can be set when provided by the user. The pure dephasing time due to a noise channel labeled λ (away from sweet spots[4]) is given by [12, 16]

$$T_\phi = A_\lambda \frac{\partial \omega_{01}}{\partial \lambda} \sqrt{2|\ln \omega_{\text{low}} t_{\text{exp}}|} \tag{3}$$

with t_{exp} representing the measurement time, and ω_{low} the low-frequency cutoff. (If not provided by the user then defaults of $t_{\text{exp}} = 10\,\mu$s and $\omega_{\text{low}}/2\pi = 1\,$Hz are used.)

As already hinted above, some qubits provide predefined methods for estimating effects due to $1/f$ flux, charge as well as Josephson junction critical-current noise channels, named `tphi_1_over_f_flux`, `tphi_1_over_f_charge` and `tphi_1_over_f_cc` respectively. Each qubit also implements a more general method `tphi_1_over_f` which accepts a user-defined operator $\partial_\lambda \hat{H}$ (for an arbitrary, user-defined noise channel λ), that is then internally used by scqubits to implement Eq. 3.

5.2 Depolarization

Noise may also cause depolarization of the qubit by inducing spontaneous transitions among eigenstates. scqubits uses the standard perturbative approach (Fermi's Golden Rule) to approximate the resulting transition rates due to different noise channels. The rate of a transition from state i to state j can be expressed as

$$\Gamma_{ij} = \frac{1}{\hbar^2} |\langle i|\hat{B}_\lambda|j\rangle|^2 S(\omega_{ij}), \tag{4}$$

where $\hat{B}_\lambda$ is the noise operator, and $S(\omega_{ij})$ the spectral density function evaluated at the angular frequency associated with the transition frequency,

[4] Currently scqubits returns a value of `numpy.inf` at sweet spots. Higher order corrections will be added at a later time.

$\omega_{ij} = \omega_j - \omega_i$. ω_{ij}, which is positive in the case of decay (the qubit emits energy to the bath), and negative in case of excitations (the qubit absorbs energy from the bath). Unless stated otherwise (see channel-specific documentation [2]), it is assumed that the depolarizing noise channels satisfy detailed balance, which implies

$$\frac{S(\omega)}{S(-\omega)} = \exp\left(\frac{\hbar\omega}{k_B T}\right), \tag{5}$$

where T is the bath temperature, and k_B Boltzmann's constant. By default, all `t1` methods estimate the depolarization times based on the *sum of the upward and downward rates*. This behavior is controlled by the argument `total`, which can be modified by the user. For example, setting `total=False` will calculate only a single-directional transition rate from the state indexed i to the state indexed j.

As in the case of $1/f$ dephasing, each qubit implements a subset of predefined depolarization methods such as `t1_capacitive` or `t1_flux_bias_line` for example (see [2] for details on what methods are implemented for each qubit). Coherence times due to arbitrary depolarization processes can also be readily calculated. This is done using a general method `t1`, which accepts an arbitrary, user-constructed $\hat{B}_\lambda$ operator, as well as a custom spectral density function $S(\omega)$.

5.3 Effective coherence times

Coherence times observed in experiments will typically be due to the combined effect of multiple contributing noise channels. scqubits can easily combine channels and compute effective coherence times or rates. In the case of depolarization, the effective coherence time is obtained from

$$\frac{1}{T_1^{\text{eff}}} = \sum_k \frac{1}{T_1^k}, \tag{6}$$

where the sum runs over all default noise channels, i.e., those methods with names beginning with `t1`). For each qubit, the included default channels can be listed by calling the qubit's `effective_noise_channels` method. To calculate the effective T_1^{eff} time based on the default noise channels and their parameters, one simply executes

```
tmon.t1_effective()
```

Similarly, scqubits can calculate an effective dephasing time T_2^{eff} (using the method `t2_effective`). This time scale includes contributions from both pure dephasing as well as depolarization and is defined as

$$\frac{1}{T_2^{\text{eff}}} = \sum_k \frac{1}{T_\phi^k} + \frac{1}{2}\sum_k \frac{1}{T_1^k}. \tag{7}$$

Once again the k index cycles over the set of default noise channels.

Both `t1_effective` as well as `t2_effective` can be easily customized, so that only select noise channels are incorporated, or calculations be based on specific user-defined parameters. For example, a smaller set of channels can be specified by passing a list of channel methods. Further, options shared by all noise channels can be set via the `common_noise_options` keyword argument, which accepts a dictionary of options. This is illustrated in the following example, where the temperature is set to $T = 0.050\,\text{K}$:

```
tmon.t1_effective(
    noise_channels=['t1_charge_impedance',
                    't1_flux_bias_line'],
    common_noise_options=dict(T=0.050)
)
```

In addition to `common_noise_options`, channel-specific noise options can be provided. This is accomplished by replacing the name of the noise-channel method with a tuple of the form `(channel_name, noise_options)` with `noise_options` a Python dictionary. In the example below, we calculate an effective T_2^{eff}, using a non-default value for the $1/f$ flux-noise strength `A_flux` that internally gets passed to the qubit's `tphi_1_over_f_flux` method:

```
tmon.t2_effective(
    noise_channels=['t1_flux_bias_line',
                    't1_capacitive',
                    ('tphi_1_over_f_flux',
                     dict(A_noise=3e-6))],
    common_noise_options=dict(T=0.050)
)
```

Table 1: List of predefined methods for estimating coherence times due to depolarization (T_1) or pure dephasing (T_ϕ). Different subsets of these are implemented for each qubit class. (See the API documentation [1] for how to change default behavior.)

Method name	Description
T_1 DEPOLARIZATION PROCESSES	
`t1_capacitive`	Capacitive loss due to dielectric dissipation [22, 24]
`t1_charge_impedance`	Loss due to charge-coupling to an impedance (e.g., open transmission line) [13, 23]
`t1_flux_bias_line`	Loss due to current fluctuations in the flux-bias line [11, 16]
`t1_inductive`	Inductive loss due to quasiparticle tunneling in Josephson junction chains that are used to implement superinductances [22, 24]
`t1_quasiparticle_tunneling`	Loss due to quasiparticle tunneling across a single Josephson junction [22, 24]
T_ϕ PURE-DEPHASING PROCESSES	
`tphi_1_over_f_cc`	Dephasing due to $1/f$ critical-current noise (fluctuations of E_J in a Josephson junction) [11, 13, 16]
`tphi_1_over_f_charge`	Dephasing due to $1/f$ charge noise (fluctuations in charge offset) [11, 13, 16]
`tphi_1_over_f_flux`	Dephasing due to $1/f$ flux noise (fluctuations in the applied magnetic flux) [11, 13, 16]

5.4 Coherence visualization

A common way to understand and visualize how noise affects a given qubit, is to plot decoherence times as a function of one of the external parameters, such as flux, charge or one of the qubit internal energy parameters, say E_J, for example. Each qubit provides a flexible method called `plot_coherence_vs_paramvals`, which facilitates this functionality. To provide an overview of the dependence of decoherence properties on, say, flux, the effect of all supported noise channels (as defined by each quibit's `supported_noise_channels` method), can be visualized in individual plots as follows:

```
tmon.plot_coherence_vs_paramvals(
    param_name='flux',
    param_vals=np.linspace(-0.5, 0.5, 100));
```

Here, `param_vals` is an array of flux values for which coherence data is generated and plotted.

The set of plots to be included can be determined by the user and further customized by either passing various plot options directly to the `plot_coherence_vs_paramvals` method, or by manipulating the properties of the matplotlib `Axes` object returned by `plot_coherence_vs_paramvals`:

```
fig, ax = tmon.plot_coherence_vs_paramvals(
    param_name='flux',
    param_vals=np.linspace(-0.5, 0.5, 100),
    noise_channels=['tphi_1_over_f_flux',
                    't1_capacitive'],
    scale=1e-3,
    color='red',
    ylabel=r"$\mu$s"
)
ax[0].set_title(r"$t_{\phi}$ from flux noise");
ax[0].set_ylim(None, 5e2)
ax[1].set_title(r"$t_1$ from capacitive loss");
```

The resulting graphical output (given a `tmon` object, as defined in Sec. 2), is shown in Fig. 5. In this example, only plots for noise channels `tphi_1_over_f_flux` and `t1_capacitive` are included. The optional `scale` argument defines a custom y-axis scale factor. Note that the default units of GHz will automatically yield coherence times in units

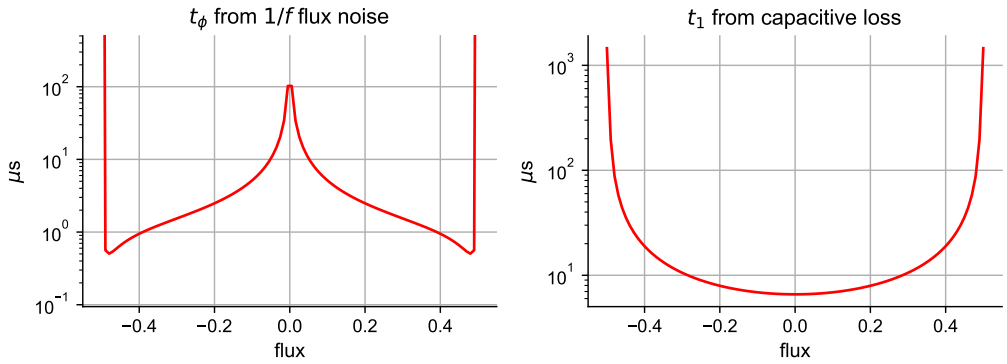


Figure 5: Customized plots showing estimated coherence times of a transmon qubit due to noise from two particular noise channels: `tphi_1_over_f_flux` and `t1_capacitive`. Since scqubits plotting routines return standard `matplotlib` `Figure` and `Axes` objects, the plot appearance can be easily modified by the user.

of ns. Setting `scale=1e-3` and adjusting the `ylabel` of the plot allows us to switch to μs on the fly. Customizing plots in such ways helps making plots visually appealing and publication-ready, without globally changing unit setting (see section 2.4).

Finally, scqubits streamlines visualization of the effective noise introduced in section 5.3 above through the methods `plot_t1_effective_vs_paramvals` and `plot_t2_effective_vs_paramvals`. Both methods work in a way analogous to `plot_coherence_vs_paramvals`, except that now a single plot of the effective coherence time is presented. Provision of an optional `noise_channels` list now sets the specific noise channels included in the calculation of T_1^{eff} and T_2^{eff}.

6 Interactive exploration

Exploring the properties of coupled quantum systems can benefit from visual aides, such as inspection how observables change when system parameters are modified. The `Explorer` class in scqubits provides multiple interactive viewgraphs collecting an important set of information regarding the user-defined system of interest.

The `Explorer` is based on a `HilbertSpace` object describing the composite circuit-QED system of interest. As explained in Section 4, `ParameterSweep` can then be used to record a sweep of an external tuning parameter. This could be, for example, an external magnetic flux, an offset charge, or even a circuit parameter such as a capacitance (difficult to change in-situ in an experiment).

6.1 Example: fluxonium coupled to a resonator

As a concrete example, we consider a system composed of a fluxonium qubit, coupled through its charge operator to the voltage inside a resonator. The initialization of the composite Hilbert space proceeds as usual: we first define the individual two subsystems that will make up the Hilbert space,

```
fluxonium = scq.Fluxonium(
    EJ=2.55,
    EC=0.72,
    EL=0.12,
    flux=0.0,
    cutoff=110,
    truncated_dim=9
)

osc = scq.Oscillator(E_osc=4.0, truncated_dim=5)
```

Here, the `truncated_dim` parameters are for the hierarchical diagonalization of the composite Hilbert space. For the fluxonium subsystem, `cutoff` fixes the internal Hilbert space dimension to 110. Once diagonalized, however, only a few eigenstates are usually meant to be retained and included in the composite Hilbert space description. In the example above, the lowest nine states are selected. Similarly, we retain five levels of the resonator, i.e., photon states $n = 0, 1, \ldots, 4$ are included.

Next, the two subsystems are declared as the two components of a joint Hilbert space:

```
hilbertspace = scq.HilbertSpace([fluxonium, osc])
```

The interaction between fluxonium and resonator is of the form $\hat{H}_{\text{int}} = g\hat{n}(a + a^\dagger)$, where $\hat{n}$ is the fluxonium's charge operator, `fluxonium.n_operator`:

```
hilbertspace.add_interaction(
    g_strength=0.2,
    op1=fluxonium.n_operator,
    op2=osc.creation_operator,
    add_hc=True
)
```

As a parameter sweep of common interest, we consider varying the external flux through the fluxonium loop. We create the necessary `ParameterSweep` object as discussed in Section 4:

```
param_name = r'$\Phi_{ext}/\Phi_0$'
param_vals = np.linspace(-0.5, 0.5, 101)

subsys_update_list = [fluxonium]

def update_hilbertspace(param_val):
    fluxonium.flux = param_val

sweep = scq.ParameterSweep(
    paramvals_by_name={param_name: param_vals},
    evals_count=10,
    hilbertspace=hilbertspace,
    subsys_update_info={param_name: [fluxonium]},
    update_hilbertspace=update_hilbertspace,
)
```

At this point, we may start the interactive `Explorer` class; sample output is shown in Fig. 6.

```
explorer = scq.Explorer(
  sweep=sweep,
  evals_count=10
)
explorer.interact()
```

7 Online presence

scqubits is an open source package, and all of its source code is freely available online under the BSD-3 license. The package is divided into three separate repositories: the first contains the source code of the core package, the second the Sphinx code for the online documentation, and finally the third collects example jupyter notebooks that illustrate how various features of scqubits can be used. To install scqubits, users can either clone the main github repository directly and install via `pip install .`, or alternatively download and install the package through the PyPI or Anaconda package repositories (see [2]). For a list of online links to the various github pages containing all the source code, online documentation, live example jupyter notebooks, as well as PyPi and Anaconda package index repository pages, see table 2. Users, are welcome and encouraged to file bug reports, post comments and suggestions, as well as initiate pull requests on the relevant github pages. Finally, users who find scqubits useful, are encouraged to cite this paper. The relevant information can be readily accessed by executing the `cite` function, like so:

```
scq.cite()
```

Table 2: scqubits web links to source code, online documentation as well as notebook examples.

https://github.com/scqubits/scqubits – github repository for scqubits package source code
https://github.com/scqubits/scqubits-doc – github repository for scqubits documentation (Sphinx source code)
https://github.com/scqubits/scqubits-examples – scqubits example jupyter notebooks on github
https://scqubits.readthedocs.io/en/latest – scqubits online documentation
https://mybinder.org/v2/gh/scqubits/scqubits-examples/released – scqubits example jupyter notebooks live demo
https://pypi.org/project/qutip/ – scqubits PyPI package repository scqubits page
https://anaconda.org/conda-forge/scqubits – scqubits Anaconda package repository scqubits page

8 Conclusions

With this paper, we have introduced the scqubits library: an open-source Python toolbox enabling the simulation of superconducting qubits – both at the single-qubit level, and at the level of composite quantum systems consisting of multiple qubits and resonators. The current functionality of the library encompasses a broad range of computational

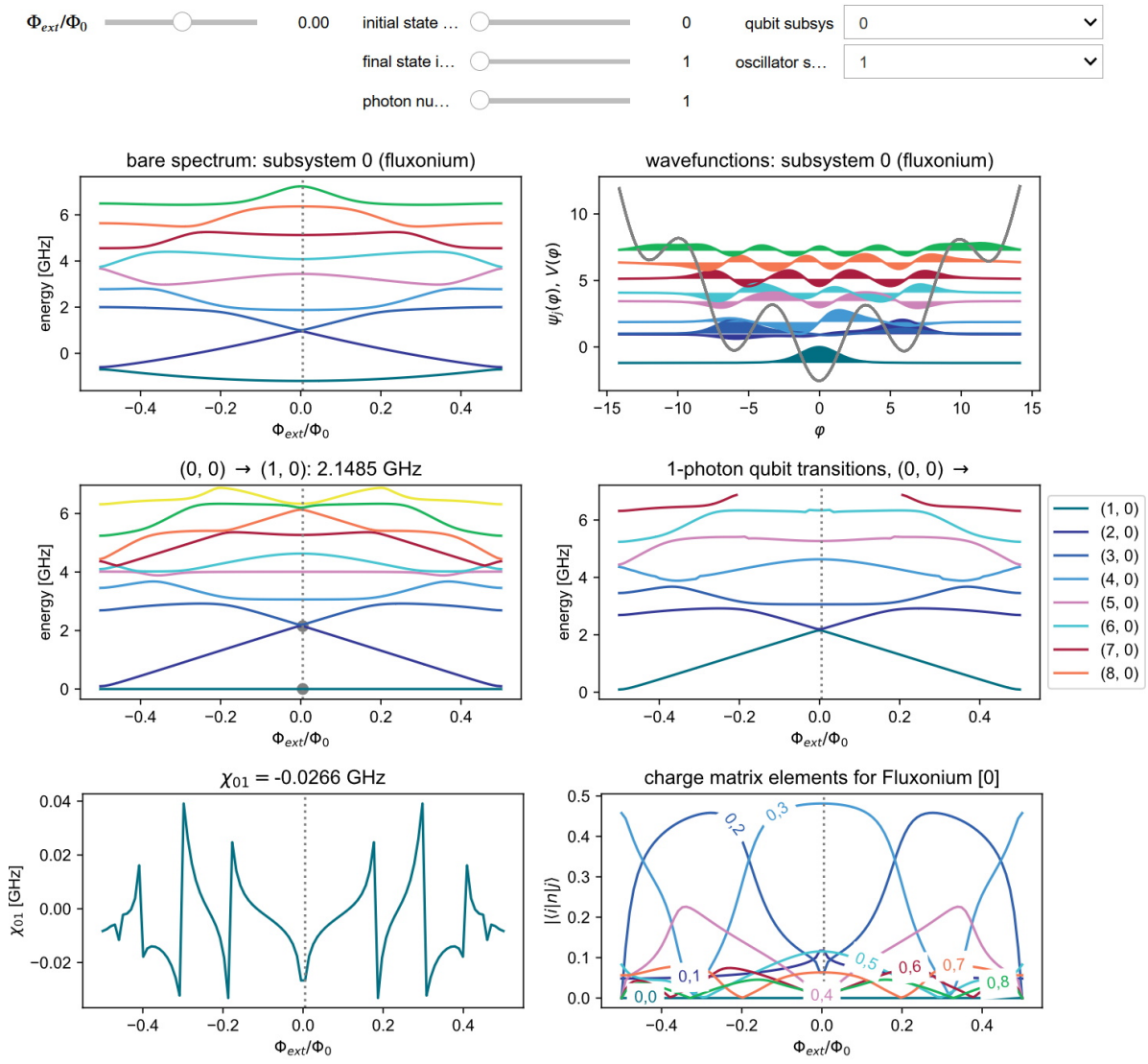


Figure 6: Example output from the interactive `Explorer` class. The plots shown are (left to right, top to bottom): 1) Bare spectra of the individual qubits, 2) Wavefunctions of the bare qubits, 3) Dressed spectrum of the composite Hilbert space, 4) Spectrum for n-photon qubit transitions, starting from a given initial state, 5) AC Stark shift χ_{01} for any of the qubits, and 6) Charge matrix elements for any of the qubits, using the same initial state as in point 4)

tasks and visualization tools commonly used in research involving superconducting qubits. While maintaining the existing interface, future work will aim to extend the scope of the library, for example by including a systematic workflow for fitting experimental two-tone spectroscopy data, analyzing custom circuits defined by the user, implementing newer qubit designs, as well adding to the list of predefined noise channels in order to extend coherence time estimations.

Acknowledgments

We thank P. Aumann, E. Blackwell, S. Chakram, F. Hassani, Z. Huang, N. Irons, P. Mundada, D. Schuster, J. Sung, S. Wang, D. Weiss, X. You, and A. Zheng for code contributions and bug reports. Continuing development of scqubits is currently supported by the AFOSR under grant FA9550-20-1-0271. Initial work on the package was in part supported by the ARO under grants W911NF-15-1-0421 and W911NF-19-1-0016, and by the Northwestern-Fermilab Center for Applied Physics and Superconducting Technologies (CAPST).

References

[1] The up-to-date API documentation can be found online. URL https://scqubits.readthedocs.io/en/latest/api-doc/apidoc.html.

[2] The full documentation for scqubits is located at the following address. URL https://scqubits.readthedocs.io/en/latest.

[3] Alexandre Blais, Ren-Shou Huang, Andreas Wallraff, S. M. Girvin, and R. J. Schoelkopf. Cavity quantum electrodynamics for superconducting electrical circuits: An architecture for quantum computation. *Phys. Rev. A*, 69:62320, 2004. DOI: 10.1103/PhysRevA.69.062320.

[4] Alexandre Blais, Arne L. Grimsmo, S. M. Girvin, and Andreas Wallraff. Circuit quantum electrodynamics. *Rev. Mod. Phys.*, 93:025005, 2021. DOI: 10.1103/RevModPhys.93.025005.

[5] Peter Brooks, Alexei Kitaev, and John Preskill. Protected gates for superconducting qubits. *Phys. Rev. A*, 87:52306, 2013. DOI: 10.1103/PhysRevA.87.052306.

[6] Guido Burkard, Roger H Koch, and D. P. DiVincenzo. Multilevel quantum description of decoherence in superconducting qubits. *Phys. Rev. B*, 69:64503, 2004. DOI: 10.1103/PhysRevB.69.064503.

[7] John Clarke and F. K. Wilhelm. Superconducting quantum bits. *Nature*, 453:1031–1042, 2008. DOI: 10.1038/nature07128.

[8] Joshua M. Dempster, Bo Fu, David G. Ferguson, D. I. Schuster, and Jens Koch. Understanding degenerate ground states of a protected quantum circuit in the presence of disorder. *Phys. Rev. B*, 90:94518, 2014. DOI: 10.1103/PhysRevB.90.094518.

[9] M. H. Devoret. Quantum fluctuations in electrical circuits. In S Reynaud, E Giacobino, and J Zinn-Justin, editors, *Quantum Fluctuations, Les Houches, Session LXIII*, chapter 10. Elsevier Science, 1995.

[10] M. H. Devoret and J. M. Martinis. Implementing qubits with superconducting integrated circuits. *Quantum Inf. Process.*, 3:163–203, 2004. DOI: 10.1007/s11128-004-3101-5.

[11] Peter Groszkowski, A. Di Paolo, A. L. Grimsmo, A. Blais, D. I. Schuster, A. A. Houck, and Jens Koch. Coherence properties of the 0-pi qubit. *New J. Phys.*, 20:043053, 2018. DOI: 10.1088/1367-2630/aab7cd.

[12] G. Ithier, E. Collin, P. Joyez, P. J. Meeson, D. Vion, D. Esteve, F. Chiarello, A. Shnirman, Y. Makhlin, J. Schriefl, and G. Schon. Decoherence in a superconducting quantum bit circuit. *Phys. Rev. B*, 72:134519, 2005. DOI: 10.1103/PhysRevB.72.134519.

[13] G. Ithier, E. Collin, P. Joyez, P J Meeson, D. Vion, D. Esteve, F. Chiarello, A. Shnirman, Y. Makhlin, J. Schriefl, and G. Schön. Decoherence in a superconducting quantum bit circuit. *Phys. Rev. B*, 72:134519, 2005. DOI: 10.1103/PhysRevB.72.134519.

[14] J. R. Johansson, P. D. Nation, and Franco Nori. QuTiP: An open-source Python framework for the dynamics of open quantum sys-

tems. *Comput. Phys. Commun.*, 183:1760–1772, 2012. DOI: 10.1016/j.cpc.2012.02.021.

[15] J. R. Johansson, P. D. Nation, and Franco Nori. QuTiP 2: A Python framework for the dynamics of open quantum systems. *Comput. Phys. Commun.*, 184:1234–1240, 2013. DOI: 10.1016/j.cpc.2012.11.019.

[16] Jens Koch, Terri M Yu, J. M. Gambetta, A. A. Houck, D. I. Schuster, J. Majer, Alexandre Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf. Charge-insensitive qubit design derived from the Cooper pair box. *Phys. Rev. A*, 76:42319, 2007. DOI: 10.1103/PhysRevA.76.042319.

[17] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver. A quantum engineer's guide to superconducting qubits. *Appl. Phys. Rev.*, 6:021318, 2019. DOI: 10.1063/1.5089550.

[18] V. E. Manucharyan, Jens Koch, L. I. Glazman, and M. H. Devoret. Fluxonium: Single Cooper-Pair Circuit Free of Charge Offsets. *Science*, 326:113–116, 2009. DOI: 10.1126/science.1175552.

[19] J. E. Mooij, T. P. Orlando, L. Levitov, Lin Tian, Caspar H. C.H. H Van der Wal, and Seth Lloyd. Josephson persistent-current qubit. *Science*, 285(5430):1036, 1999. DOI: 10.1126/science.285.5430.1036.

[20] Y. Nakamura, Yu. A. Pashkin, and J. S. Tsai. Coherent control of macroscopic quantum states in a single-Cooper-pair box. *Nature*, 398:786–788, 1999. DOI: 10.1038/19718.

[21] T. P. Orlando, J. E. Mooij, Lin Tian, Caspar H. Van Der Wal, L. S. Levitov, Seth Lloyd, and J. J. Mazo. Superconducting persistent-current qubit. *Phys. Rev. B*, 60:15398, 1999. DOI: 10.1103/PhysRevB.60.15398.

[22] Ioan M. Pop, Kurtis Geerlings, G. Catelani, Robert J. Schoelkopf, L. I. Glazman, and M. H. Devoret. Coherent suppression of electromagnetic dissipation due to superconducting quasiparticles. *Nature*, 508(7496):369–72, 2014. DOI: 10.1038/nature13017.

[23] R. J. Schoelkopf, A. A. Clerk, S. M. Girvin, K. W. Lehnert, and M. H. Devoret. Qubits as Spectrometers of Quantum Noise. 5115:1–31, 2002. DOI: 10.1007/978-94-010-0089-5_9.

[24] W. C. Smith, A. Kou, X. Xiao, U. Vool, and M. H. Devoret. Superconducting circuit protected by two-Cooper-pair tunneling. *npj Quantum Inf.*, 6:1–9, 2020. DOI: 10.1038/s41534-019-0231-2.

[25] Uri Vool and Michel Devoret. Introduction to quantum electromagnetic circuits. *Int. J. Circuit Theory Appl.*, 45:897–934, 2017. DOI: 10.1002/cta.2359.

[26] A. Wallraff, D. I. Schuster, Alexandre Blais, L. Frunzio, R. S. Huang, J. Majer, S. Kumar, S. M. Girvin, and R. J. Schoelkopf. Strong coupling of a single photon to a superconducting qubit using circuit quantum electrodynamics. *Nature*, 431:162–167, 2004. DOI: 10.1038/nature02851.

[27] J. Q. You and Franco Nori. Superconducting circuits and quantum information. *Phys. Today*, 58:42–47, 2005. DOI: 10.1063/1.2155757.

Appendix

Superconducting qubit and oscillator classes

scqubits currently implements several common types of superconducting qubits along with a linear and non-linear oscillators, as well as a generic qubit (i.e., a simple two-level system), each realized as a Python class[5]. A brief summary is shown in the following table [2]:

class	description
`Transmon`, `TunableTransmon`	Transmon qubit or Cooper pair box [16, 20]
`Fluxonium`	fluxonium qubit [18]
`FluxQubit`	3-junction flux qubit [19]
`ZeroPi`	0-π qubit (symmetric) [5, 8]
`FullZeroPi`	0-π qubit (coupled to ζ-mode) [8]
`Cos2PhiQubit`	$\cos 2\phi$ qubit [24]
`Oscillator`	Quantum harmonic oscillator
`KerrOscillator`	Nonlinear Kerr oscillator
`GenericQubit`	A two-level system

All the qubit classes (except for `GenericQubit`) define a number of important methods that can be used for diagonalization, computation of matrix elements and spectral data, as well as for plotting. A few of these are summarized in Table 3.

Besides these methods, each superconducting qubit class also implements a predefined number of quantum operators which can simplify doing various calculations. These may include the phase $\hat{\phi}$ or number $\hat{n}$ operators, but also more specialized ones such as $\cos(\hat{\phi})$ and the like. See the API documentation [1], for a comprehensive list.

In the following sections we describe the qubit and oscillator classes that scqubits implements in more detail: present their circuit diagrams (where applicable), give definitions of the respective Hamiltonians, and briefly discuss their numerical implementation in the library.

[5]More qubit types will be added in the future. See online documentation for the latest information [2].

A.I Harmonic and Kerr oscillators

The most basic quantum systems that can be realized in scqubits are harmonic and Kerr resonators with Hamiltonians

$$\hat{H}_{\text{osc}} = E_{\text{osc}}\hat{a}^\dagger\hat{a}, \tag{A.1}$$

and

$$\hat{H}_{\text{Kerr}} = E_{\text{osc}}\hat{a}^\dagger\hat{a} - K\hat{a}^\dagger\hat{a}^\dagger\hat{a}\hat{a}, \tag{A.2}$$

respectively, where $\hat{a}$ corresponds to a standard bosonic lowering operator, while E_{osc} and K are the oscillator and Kerr energies, respectively. Example initialization code for both cases is shown below. For a harmonic oscillator, we have

```
osc=scq.Oscillator(E_osc=5)
```

while for the Kerr oscillator,

```
kerr=scq.KerrOscillator(E_osc=5, K=0.1,
                        l_osc=0.1)
```

Note that the oscillator length `l_osc` is an optional parameter in both oscillator classes, which if not given, is set to `None`. It, however, has to be provided by the user for the oscillator classes to define phase $\hat{\phi} = l_{\text{osc}}(\hat{a}^\dagger + \hat{a})/\sqrt{2}$ and number $\hat{n} = i(\hat{a}^\dagger - \hat{a})/\sqrt{2}l_{\text{osc}}$ operators[6], which are implemented in methods `phi_operators` as well as `n_operator` respectively.

A.II Generic Qubit

scqubits implements a generic qubit class called `GenericQubit`, that corresponds to a simple two-level system with a Hamiltonian

$$\hat{H} = \frac{E}{2}\hat{\sigma}_z, \tag{A.3}$$

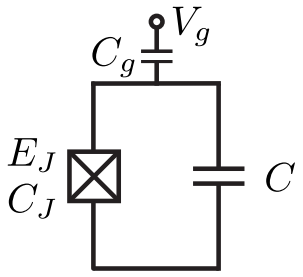
where E is the qubit's energy. This class implements a set of Pauli operators $\hat{\sigma}_k$, with $k = \{x, y, z\}$, as well as lowering and raising operators $\hat{\sigma}_\pm$, with methods names such as `sx_operator`, `sm_operator`, etc.. To initialize a `GenericQubit` object, one provides its energy:

```
qubit = scq.GenericQubit(E=5.5)
```

[6]For a quantum harmonic oscillator represented by a Hamiltonian $H = E_{\text{kin}}\hat{P}^2 + E_{\text{pot}}\hat{X}^2$, the oscillator length is defined as $l_{\text{osc}} = (E_{\text{kin}}/E_{\text{pot}})^{1/4}$.

A.III Transmon qubit

The Cooper pair box [20] and transmon qubit [16] share the same underlying circuit composed of a single Josephson junction and a parallel capacitance which may either be the pure junction capacitance, or include an external shunt capacitance:



The circuit Hamiltonian can be written as

$$\hat{H}_{\text{transmon}} = 4E_C(\hat{n} - n_g)^2 - E_J \cos\hat{\phi}, \qquad \text{(A.4)}$$

where $\hat{n}$ ($\hat{\phi}$) is the charge (phase) operator[7] Here $E_C = e^2/2C_\Sigma$ is the charging energy associated with the combined capacitances of the junction, the shunt capacitor, and any additional ground capacitance and/or capacitance to a charge bias line, $C_\Sigma = C_J + C + C_g$. The Josephson energy of the junction is related to its critical current via $E_J = I_c\Phi_0/2\pi$. The quantity n_g is the dimensionless offset charge capturing the capacitive coupling to a bias voltage source as well as electric-potential fluctuations of the environment.

Internally, the `Transmon` class employs the charge-basis representation to construct the Hamiltonian matrix. This matrix is infinite-dimensional, in principle, and must hence be truncated. To this end, a charge-number cutoff `ncut` is introduced. Given this cutoff, the included charge states $|n\rangle$ are within the range `-ncut`$\leq n \leq$`ncut`. The cutoff must be chosen sufficiently large to avoid truncation errors[8].

[7]Since the Hamiltonian is periodic in $\hat{\phi}$, we stress that only periodic functions of $\hat{\phi}$ are formally defined.

[8]For low-lying transmon wavefunctions ($E_J/E_C \gg 1$), there is a simple cutoff criterion: The ground state is close to Gaussian with standard deviation $\sigma = (8E_C/E_J)^{1/4}$. Treating n as continuous and assuming an n_g of order 1, Fourier transform of the ground state yields $\psi_0(n)$ which is also close to a Gaussian, here with standard deviation $\sigma' = (E_J/8E_C)^{1/4}$. Using a 3σ estimate, one finds that `ncut` should be no smaller than $\texttt{ncut}_{\min} \approx \lceil 2(E_J/E_C)^{1/4} \rceil$.

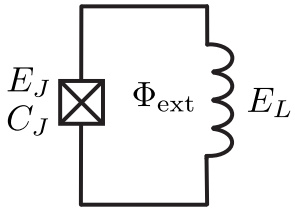
[t]

An example initialization code of a Transmon qubit is shown below

```
transmon = scq.Transmon(EJ=30.02, EC=0.2,
                        ng=0.0, ncut=101)
```

A.IV Fluxonium qubit

The circuit of the fluxonium qubit [18] consists of a Josephson junction shunted by a large inductor:



The resulting qubit Hamiltonian takes the form

$$\hat{H}_{\text{fluxonium}} = 4E_C\hat{n}^2 - E_J\cos(\hat{\varphi} - \varphi_{\text{ext}}) + \frac{1}{2}E_L\hat{\varphi}^2, \qquad \text{(A.5)}$$

where $E_C = e^2/2C_J$ and E_J are the charging and Josephson energies of the junction, and $E_L = (\frac{\Phi_0}{2\pi})^2/L$ is the inductive energy. The Hilbert-space basis used by scqubits for construction of the Hamiltonian is the harmonic-oscillator basis associated with the inductor and junction capacitor. The Hamiltonian can be rewritten in this basis by employing the usual ladder operators a, $a^\dagger$,

$$\hat{H}_{\text{fluxonium}} = \sqrt{8E_LE_C}\,a^\dagger a - \frac{E_J}{2}e^{-i\varphi_{\text{ext}}}\exp\left[\frac{i\varphi_{\text{osc}}}{\sqrt{2}}(a + a^\dagger)\right] + \text{h.c.} \qquad \text{(A.6)}$$

Here, we have rewritten $\cos(\hat{\varphi} - \varphi_{\text{ext}}) = \frac{1}{2}e^{-i\varphi_{\text{ext}}}e^{i\hat{\varphi}} + \text{h.c.}$, and used $\hat{\varphi} = \frac{\varphi_{\text{osc}}}{\sqrt{2}}(\hat{a} + \hat{a}^\dagger)$ with $\varphi_{\text{osc}} = (8E_C/E_L)^{1/4}$ denoting the oscillator length. Numerical evaluation of the matrix exponential in (A.6) is carried out via `scipy.linalg.expm()`.

It must be noted that the harmonic-oscillator basis is not well-adapted to fluxonium eigenstates that localize in individual wells of the cosine contribution to the potential. Consequently, the inevitable truncation in the harmonic-oscillator basis must generally proceed with caution, and the cutoff number `cutoff` be chosen sufficiently large for convergence.

Table 3: A summary of a few selected methods shared by all the qubit classes. For full information including the call signatures of these methods, see the API documentation [1].

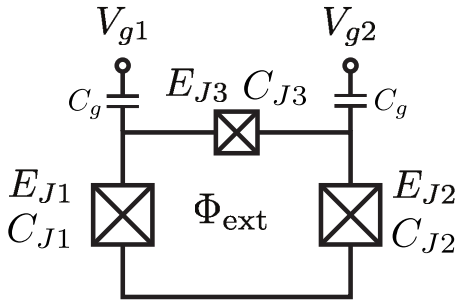
Qubit class method	description
Basics	
`hamiltonian`	Hamiltonian matrix (in basis specific to each qubit)
`eigenvals`	Eigenvalues of the qubit Hamiltonian
`eigensys`	Eigenvalues and eigenvectors of the qubit Hamiltonian
Matrix elements, Spectral data	
`wavefunction`	Wavefunction of qubit (in basis dependent on each qubit)
`matrixelement_table`	Matrix elements for a specified qubit operator, with respect to a set of qubit eigenstates
`get_spectrum_vs_paramvals`	Compute eigenenergies and eigenstates as a function of a specified external parameter
`get_matelements_vs_paramvals`	Compute matrix elements for specified qubit operator as a function of a specified external parameter
Plotting methods	
`plot_wavefunction`	Plot wavefunction(s) of qubit
`plot_evals_vs_paramvals`	Plot of energy eigenvalues as a function of specified external parameter
`plot_matrixelements`	Combined 3d bar plot and 2d plot of matrix elements for specified qubit operator
`plot_matelem_vs_paramvals`	Plot of matrix elements for specified qubit operator as a function of external parameter

An example initialization code of a Fluxonium qubit is show below

```
fluxonium = scq.Fluxonium(EJ=5.7, EC=1.2,
                          EL=1.0, cutoff = 150,
                          flux = 0.5,
                          truncated_dim=10)
```

A.V Flux qubit

The 3-junction flux qubit that scqubits implements, was first proposed in [21]. Its circuit consists of 3 Josephson junctions in a loop that is threaded with an external flux Φ_{ext}.



E_{Jk} (C_{Jk}) with $k \in \{1, 2, 3\}$ are the Josephson energies (capacitances) associated with each junction. For normal qubit operations one junction is chosen to be smaller than the other two. The effective Hamiltonian of such a flux qubit is described by

$$\hat{H}_{\text{flux}} = \sum_{j,k=1}^{2} 4(\mathsf{E}_{\text{C}})_{jk}(\hat{n}_j - n_{gj})(\hat{n}_j - n_{gk}) - \sum_{k=1}^{2} E_{Jk} \cos\hat{\varphi}_k - E_{J3}\cos(\hat{\varphi}_1 - \hat{\varphi}_2 + \varphi_{\text{ext}}). \tag{A.7}$$

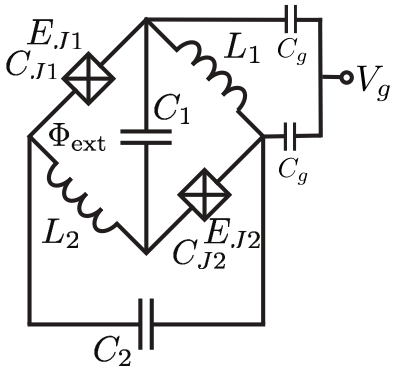
In the above expression, $\mathsf{E_C}$ represents a charging energy matrix (which includes effects of small capacitors C_g), while n_{gj} (with $j \in \{1, 2\}$) are the charge offsets. The two degrees of freedom are represented by $\hat{\varphi}_1$ and $\hat{\varphi}_2$, with their conjugates charge operators $\hat{n}_1$ and $\hat{n}_2$ respectively. Numerical diagonalization is performed in the charge basis for both $\hat{\varphi}_1$ and $\hat{\varphi}_2$ (see discussion in A.III).

A sample initialization code of the Flux qubit, where the third junction is assumed to be smaller than the other two by a factor of α, is shown below

```
EJ = 35.0
alpha = 0.6
fluxqubit = scq.FluxQubit(
    EJ1 = EJ,
    EJ2 = EJ,
    EJ3 = alpha*EJ,
    ECJ1 = 1.0,
    ECJ2 = 1.0,
    ECJ3 = 1.0/alpha,
    ECg1 = 50.0,
    ECg2 = 50.0,
    ng1 = 0.0,
    ng2 = 0.0,
    flux = 0.5,
    ncut = 10
)
```

A.VI 0-π qubit

The $0 - \pi$ qubit was first proposed in Ref. [5]. Its circuit consists of two Josephson junctions, two capacitors as well as two superinductors arranged in the following form:



The behavior of this qubit has been shown to strongly depend on the presence of parameter disorder [8, 11]. In particular, when the two (non-junction) capacitive or inductive energies are not identical (i.e., $C_1 \neq C_2$ or $L_1 \neq L_2$), the core qubit degrees of freedom $\hat{\theta}$ and $\hat{\phi}$ end up coupling to a low-frequency harmonic mode $\hat{\zeta}$, which would stay uncoupled when there is no disorder in in these parameters. For this reason, scqubits implements two separate classes that can be used for modeling of $0 - \pi$ qubits. The first, `ZeroPi`, assumes both of the inductors as well as the (non-junction) capacitors are identical and hence only includes the $\hat{\theta}$ and $\hat{\phi}$ degrees of freedom. The second, `FullZeroPi`, allows for small parameter disorder in the superinductnaces L_k as well as the (non-junction) capacitors C_k, resulting in a three degrees of freedom system, which now also includes the harmonic $\hat{\zeta}$ mode. Note that the `ZeroPi` class does allow disorder in the Josephson junctions (i.e., E_{J1} and C_{J1} do not have to be identical to E_{J2} and C_{J2}), as this kind of disorder still leaves the $\hat{\zeta}$ mode decoupled from $\hat{\theta}$ and $\hat{\phi}$.

To quantify disorder in any parameter X (with $X \in \{C, E_L, C_J, E_J\}$, we define

$$X = \frac{X_1 + X_2}{2} \qquad dX = \frac{X_1 - X_2}{X}. \tag{A.8}$$

Then, in the limit of small disorder dX [8, 11], a general $0 - \pi$ Hamiltonian can be approximated by

$$\hat{H}_{\text{tot}} = \hat{H}_{0-\pi} + \hat{H}_{\text{int}} + \hat{H}_{\zeta} \tag{A.9}$$

with

$$\begin{aligned} \hat{H}_{0-\pi} =& 2E_{\text{CJ}}\hat{n}_\phi^2 + 2E_{\text{C}\Sigma}(\hat{n}_\theta + n_g)^2 \\ & - 2E_J \cos\hat{\theta} \cos\left(\phi - \frac{\varphi_{\text{ext}}}{2}\right) + E_L\hat{\phi}^2 \\ & - 2E_{C\Sigma} dC_J\, \hat{n}_\phi \hat{n}_\theta \\ & + E_J dE_J \sin\hat{\theta} \sin\left(\hat{\phi} - \frac{\phi_{\text{ext}}}{2}\right), \end{aligned} \tag{A.10}$$

along with

$$\hat{H}_{\text{int}} = -2E_{C\Sigma} dC\, \hat{n}_\theta \hat{n}_\zeta + E_L dE_L \hat{\phi}\,\hat{\zeta}, \tag{A.11}$$

and

$$\hat{H}_{\zeta} = \omega_\zeta \hat{a}^\dagger \hat{a}. \tag{A.12}$$

The quantity n_g is the charge offset of $\hat{n}_\theta$, while $\varphi_{\text{ext}} = 2\pi\Phi_{\text{ext}}/\Phi_0$. The class `ZeroPi` only implements $\hat{H}_{0-\pi}$, as it clear from the above description, that when $dE_L = dC = 0$, $\hat{H}_{\text{int}} = 0$, , while

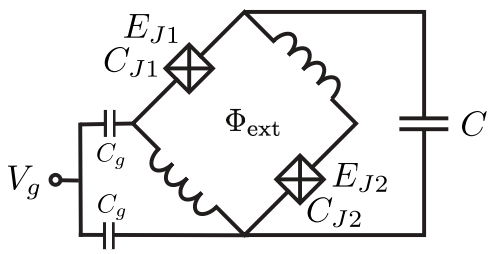
`FullZeroPi` includes all three terms of $\hat{H}_{\text{tot}}$. Internally, we use charge basis to describe the $\hat{\theta}$ degree of freedom (see A.III), phase basis for the $\hat{\phi}$ degree of freedom, and finally $\hat{\zeta}$ is modeled using harmonic basis (see A.IV). For problems where the disorder dC and dE_L can be neglected, it is strongly recommended that `ZeroPi` is used, as the performance penalty from including the physics of the $\hat{\zeta}$ mode can be substantial.

Below is sample code showing an initialization of a $0-\pi$ qubit. Here, we include a 5% disorder in the Josephson junction energies, but assume there is no disorder in the superinductors and (non-junction) capacitors, allowing us to use the `ZeroPi` class:

```
zeropi_dis = scq.ZeroPi(
    grid = phi_grid,
    EJ   = 10.0,
    dEJ = 0.05,
    EL   = 0.04,
    ECJ  = 20.0,
    dCJ = 0.05,
    EC = 0.04,
    ng   = 0.3,
    flux = 0.2,
    ncut = 30
)
```

A.VII $\cos 2\phi$ qubit

The $\cos 2\phi$ qubit was proposed in [24]. Its circuit includes a superconducting loop, threaded by an external flux Φ_{ext}, which consists of two superinductors and two Josephson junctions, with an appropriately placed shunt capacitance:



Such topology can be engineered to only allow pairs of cooper pairs to tunnel, leading a level of intrinsic protection from noise [24]. The Hamiltonian of a $\cos 2\phi$ qubit can be written as

$$\begin{aligned}\hat{H}_{\cos 2\phi} = &2E'_{CJ}\hat{n}_\phi^2 + 2E'_{CJ}(\hat{n}_\theta - n_g - \hat{n}_\zeta)^2 + 4E_C\hat{n}_\zeta^2 \\ &+ E'_L(\hat{\phi} - \pi\Phi_{\text{ext}}/\Phi_0)^2 \\ &+ E'_L\hat{\zeta}^2 - 2E_J\cos\hat{\theta}\cos\hat{\phi} \\ &+ 2dE_JE_J\sin\hat{\theta}\sin\hat{\phi} \\ &- 4dC_JE'_{CJ}\hat{n}_\phi(\hat{n}_\theta - n_g - n_\zeta) \\ &+ dLE'_L(2\hat{\phi} - \varphi_{\text{ext}})\zeta,\end{aligned} \tag{A.13}$$

with $E'_{CJ} = E_{CJ}/(1-dC_J)^2$ and $E'_L = E_L/(1-dL)^2$. Parameters of the form dX represent disorder[9]. In particular, we have

$$dX = \frac{X_1 - X_2}{X_1 + X_2}, \tag{A.14}$$

for $X \in \{L, C_J\}$, with L (C_J) being the superinductance (junction capacitance). We define the charging (inductive) energy of the Josephson junction capacitor (superinductor) as $E_{CJk} = e^2/2C_{Jk}$ ($E_{Lk} = (\frac{\Phi_0}{2\pi})^2/L_k$). These expressions let us further write

$$Y = \frac{2Y_1Y_2}{Y_1 + Y_2}, \tag{A.15}$$

with $Y \in \{E_L, E_{CJ}\}$. Finally, the Josephson energy satisfies

$$E_J = \frac{E_{J1} - E_{J2}}{E_{J1} + E_{J2}}. \tag{A.16}$$

Similarly to the $0-\pi$ qubit, $\cos 2\phi$ circuit consists of three degrees of freedom: θ, ϕ and ζ, with their respective conjugates defined as $\hat{n}_\theta$, $\hat{n}_\phi$ and $\hat{n}_\zeta$. The quantity n_g represents the charge offset of the $\hat{n}_\theta$ variable. We stress that the labels and notation utilized by scqubits for the degrees of freedom and some of the circuit parameters differs slightly from [24]. The following table outlines how to convert between the two:

[9] Note the difference in the definition of disorder here versus the definition for the $0-\pi$ qubit. The definition of parameter disorder in the $\cos 2\phi$ qubit used by scqubits follows the notation in [24].

scqubits	Reference [24]
ζ	θ
θ	φ
ϕ	$\frac{\phi}{2}$
E_C	$x\epsilon_C$
E_{CJ}	ϵ_C
E_J	ϵ_J
E_L	ϵ_L

To numerically diagonalize the Hamiltonian of the $\cos 2\phi$ qubit, the harmonic basis are used for both the $\hat{\phi}$ and $\hat{\zeta}$ variables (see the discussion of the Fluxonium qubit in A.IV for more details), while the charge basis are used for the $\hat{\theta}$ variable (see A.III). When instantiating an scqubits object corresponding to this qubit, the user needs to specify cutoffs for basis states described above (this is done using `phi_cut`, `zeta_cut`, and `ncut`), which need to be chosen large enough so that convergence is achieved.

An instance of the $\cos 2\phi$ qubit can be initialized as follows

```
cos2phi_qubit = scq.Cos2PhiQubit(
    EJ = 15.0,
    ECJ = 2.0,
    EL = 1.0,
    EC = 0.04,
    dCJ = 0.0,
    dL = 0.6,
    dEJ = 0.0,
    flux = 0.5,
    ng = 0.0,
    ncut = 7,
    phi_cut = 7,
    zeta_cut = 30
)
```