CircuitQ

Philipp Aumann, Tim Menke, William Oliver, Wolfgang Lechner

Jun 01, 2022

CONTENTS

CircuitQ is an an open-source toolbox for the analysis of superconducting circuits implemented in Python. It features the automated construction of a symbolic Hamiltonian of the input circuit, as well as a dynamic numerical representation of this Hamiltonian with a variable basis choice. Additional features include the estimation of the T1 lifetimes of the circuit states under various noise mechanisms. The circuit quantization is both applicable to circuit inputs from a large design space and open-source.

To get started, have a look at the installation instructions and at our quick demo for a demonstration of the basic usage of the toolbox.

A more comprehensive description of CircuitQ's functionalities can be found in the tutorial.

For further illustrations of the toolbox, please see the more examples section. The API reference provides an overview of the range of functions and is constructed based on the docstrings.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation

1.1.1 Installation via pip

CircuitQ can be installed by using python's package manager `pip`:

```
pip install circuitq
```

1.1.2 Installation via conda

If you are using the Anaconda distribution, you may want to avoid mixing `pip` and `conda`, due to a different handling of dependencies of those installers[1]. CircuitQ can also be installed using `conda` via the conda-forge channel. You could either first add conda-forge to your installation channels and subsequently install CircuitQ by

```
conda config --add channels conda-forge
conda install circuitq
```

or you can state the channel directly for the installation:

```
conda install -c conda-forge circuitq
```

1.1.3 Installation via GitHub

Alternatively, you can clone CircuitQ's repository at GitHub and run one of the following commands inside the projects main directory:

```
python setup.py develop
```

This is recomended for developers as it keeps the package up to date, or:

```
pip install .
```

[1] The situation improved with conda version 4.6.0 which provides the experimental `pip_interop_enabled` feature. However best practice is still to avoid mixing conda and pip if possible.

1.2 Quick Demo

This notebook is a brief domonstration of the usage of CircuitQ for the Transmon as an exemplary circuit.

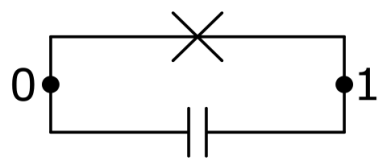
Please refer to the Tutorial section for more details.

[1]:
```
import circuitq as cq

import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
```

1.2.1 Exemplary Circuit: Transmon

Create circuit



[2]:
```
graph = nx.MultiGraph()
graph.add_edge(0,1, element = 'C')
graph.add_edge(0,1, element = 'J')

circuit = cq.CircuitQ(graph)
```

Symbolic Hamiltonian

[3]:
```
circuit.h
```

[3]:
$$-E_{J010}\cos\left(\frac{\Phi_1}{\Phi_o}\right)+\frac{0.5q_1^2}{C_{01}}$$

Parameters

[4]:
```
circuit.h_parameters
```

[4]:
```
[C_{01}, E_{J010}]
```

Numerical Hamiltonian

[5]:
```
h_num = circuit.get_numerical_hamiltonian(401, grid_length=np.pi*circuit.phi_0)
```

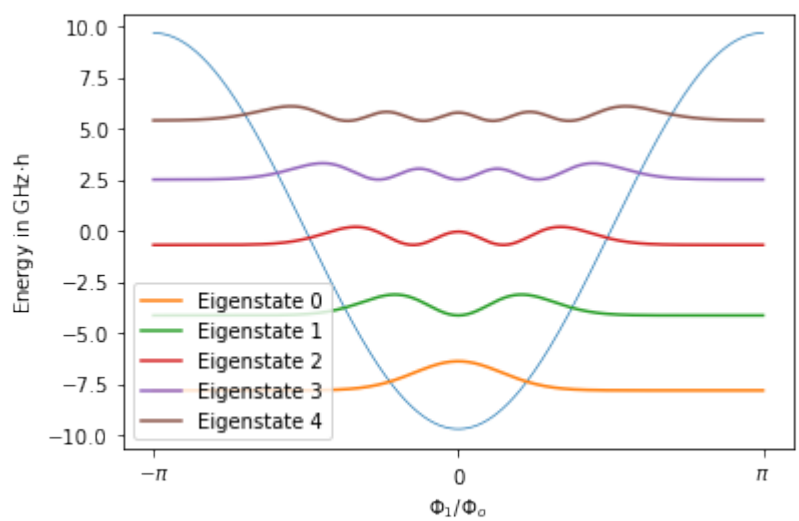
Diagonalization

[6]:
```
eigv, eigs = circuit.get_eigensystem()
```

Plot Spectrum

[7]:
```
circuit.transform_charge_to_flux()
eigs = circuit.estates_in_phi_basis
n_states = 5
state_scaling = 15 * (eigv[n_states-1]-eigv[0]/n_states)
h = 6.62607015e-34
y_scaling = 1/(h *1e9)

def potential(phi):
    return -circuit.c_v["E"]*np.cos(phi/circuit.phi_0)
plt.plot(circuit.flux_list, potential(circuit.flux_list)*y_scaling, lw=0.7)
for n in range(n_states):
    plt.plot(circuit.flux_list,
             (eigv[n] + np.real(eigs[n]*np.conjugate(eigs[n]))*state_scaling)*y_scaling,
             label="Eigenstate " +str(n))
plt.legend()
plt.xticks(np.linspace(-1*np.pi, 1*np.pi, 3)*circuit.phi_0 ,
           [r'$-\pi$',r'$0$',r'$\pi$'])
plt.xlabel(r"$\Phi_1/ \Phi_o$")
plt.ylabel(r"Energy in GHz$\cdot$h")
plt.show()
```



$T1$ Times

[8]:
```
T1_qp = circuit.get_T1_quasiparticles()
print("Quasiparticles noise contribution T1 = {:e} s".format(T1_qp))
```

```
Quasiparticles noise contribution T1 = 3.103361e-03 s
```

[9]:
```
T1_c = circuit.get_T1_dielectric_loss()
print("Charge noise contribution T1 = {:e} s".format(T1_c))
```

```
Charge noise contribution T1 = 1.836958e-04 s
```

[10]:
```
print("Total T1 = {:e} s".format( 1/( 1/T1_qp + 1/T1_c)))
```

```
Total T1 = 1.734300e-04 s
```

1.3 Tutorial

This tutorial will guide through the creation and analysis of a superconducting circuit with CircuitQ. It shows some of the core functionalities of CircuitQ and demonstrates how to use them. While the variety of possible input circuits is high, we will here focus on the Fluxonium as an example.

1.3.1 Import statements

First, import `circuitq` and `networkx`. The latter toolbox is required to construct the circuit graph.

[1]:
```
import circuitq as cq
import networkx as nx
```

It is also convenient to import `numpy` for the use of numerical expressions and `pyplot` to visualize some results.

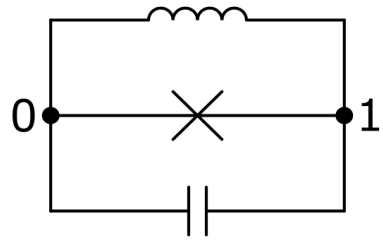
[2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.3.2 Create circuit graph

A superconducting circuit can be related to a graph by relating all circuit elements between two nodes to an edge of the graph. Possible elements are capacitances (`'C'`), linear inductances (`'L'`) and Josephson junctions (`'J'`).

Since multiple edges of such a graph can be in parallel we work with the `MultiGraph` class of `networkx`.

Let's consider the Fluxonium as an example, which consists of a linear inductance and a Josephson junction in parallel together with a shunted capacity.



To initialize the circuit, we execute the following:

[3]:
```
graph = nx.MultiGraph()
graph.add_edge(0,1, element = 'C')
graph.add_edge(0,1, element = 'J')
graph.add_edge(0,1, element = 'L')

circuit = cq.CircuitQ(graph)
```

Here, we created a graph with three elements in between node 0 and node 1. The `CircuitQ` class is the core of the toolbox. An element of this class corresponds to a superconducting circuit. Ground nodes and charge offset nodes can be specified upon initialization. Please refer to the API reference for more details.

CircuitQ's function `visualize_circuit_general()` can be used to visualize the graph.

1.3.3 Symbolic Hamiltonian

CircuitQ performs an automated quantization of the input circuit. It provides a symbolic `SymPy` expression for the corresponding circuit Hamiltonian.

[4]:
```
circuit.h
```

[4]:

$$-E_{J010}\cos\left(\frac{\Phi_1}{\Phi_o}\right)+\frac{\left(\Phi_1+\tilde{\Phi}_{010}\right)^2}{2L_{010}}+\frac{0.5q_1^2}{C_{01}}$$

1.3.4 Parameters

Besides the flux and charge variables ϕ and q, the Hamiltonian contains system parameters, whose values can be changed to tune the circuit.

[5]:
```
circuit.h_parameters
```

[5]:
```
[C_{01}, E_{J010}, L_{010}, \tilde{\Phi}_{010}]
```

In this case it contains the capacity C_{01}, the Josephson energy E_{J010} and the inductance L_{010}. During quantization, CircuitQ automatically detects inductive loops and assigns loop fluxes to them, which is represented by $\tilde{\Phi}_{010}$. If a charge offset would have been assigned upon initialization, it would also appear as a parameter qo.

Let's assign a numerical value for the Josephson energy and the external flux

[6]:
```
EJ = circuit.c_v["E"]/3
phi_ext = np.pi*circuit.phi_0
```

`circuit.c_v` is a dictionary for characteristic parameter values. `circuit.phi_0` is the flux quantum, which we define as $\Phi_o = \frac{\hbar}{2e}$.

1.3.5 Numerical Hamiltonian

To analyze the circuit's quantum physical properties, a numerical representation of the Hamiltonian has to be generated.

[7]:
```
h_num = circuit.get_numerical_hamiltonian(401,
                    parameter_values=[False, EJ, False, phi_ext ])
```

The method `get_numerical_hamiltonian()` takes the length of the matrices representing the flux and charge variables as the first parameter input (here `401`). CircuitQ automatically performs the numerical implementation of the node variables either in the charge basis, in the flux basis or also in a combination of both - depending on the periodicity of the potential along the individual flux variables. In the exemplary case of the Fluxonium, the flux basis will be used due to the harmonic contribution of the linear inductance. The length of the numerical flux coordinate grid can be set manually by using the keyword `grid_length`. This may have a crucial impact on the numerical results. If it is not specified (like in the example above) it is set to the default value $\Phi_{max} = 4\pi\Phi_o$. For the cases which are implemented in the charge basis, `CircuitQ`'s method `transform_charge_to_flux()` can be used to be able to express the eigenstates still in the flux basis (see More Examples -> Flux Qubit).

To assign specific parameter values, use an input list for `paramter_values`, with the same order as the `h_parameters` list. If a value is set to `False`, a characteristic default value will be assigned to it.

The numerical Hamiltonian is represented as a sparse `csr_matrix` from the `SciPy` package.

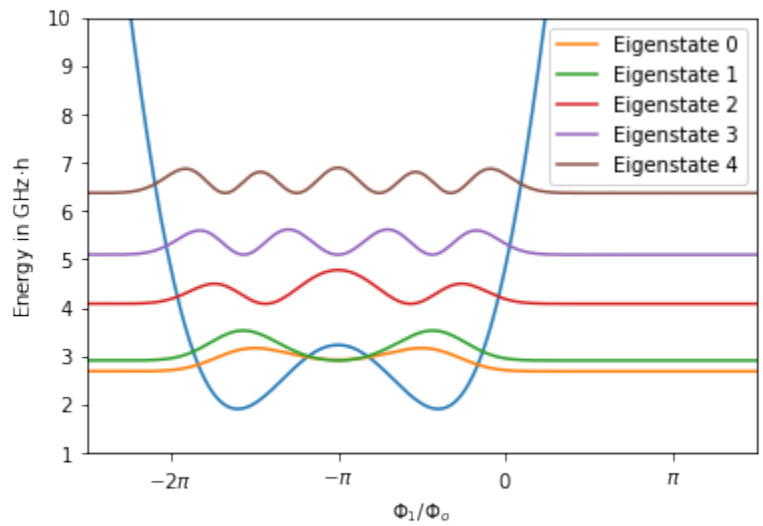
1.3.6 Diagonalization

To get the (lowest) eigenvalues and eigenstates of the numerical Hamiltonian, use `get_eigensystem()` which is a wrapper around `SciPy`'s `eigsh` function for sparse matrices.

[8]:
```
eigv, eigs = circuit.get_eigensystem()
```

Finally, we can plot the lowest eigenstates of the circuit.

[9]:
```
h = 6.62607015e-34
y_scaling = 1/(h *1e9)
plt.plot(circuit.flux_list, np.array(circuit.potential)*y_scaling, lw=1.5)
for n in range(5):
    plt.plot(circuit.flux_list,
             (eigv[n]+ abs(eigs[:,n])**2*2e-23)*y_scaling
             ,label="Eigenstate " +str(n))
plt.legend()
plt.xlabel(r"$\Phi_1 / \Phi_o$")
plt.ylabel(r"Energy in GHz$\cdot$h")
plt.xticks(np.linspace(-2*np.pi, 1*np.pi, 4)*circuit.phi_0 ,
           [r'$-2\pi$', r'$-\pi$',r'$0$',r'$\pi$'])
plt.xlim(-2.5*np.pi*circuit.phi_0, 1.5*np.pi*circuit.phi_0)
plt.ylim(1,10)
plt.show()
```



To plot the potential, the `potential` attribute of a `CircuitQ` instance can be used if the problem is 1-dimensional and formulated in flux basis.

1.3.7 Anharmonicity

To operate the circuit as a qubit, a certain degree of the spectrum's anharmonicity is desired. CircuitQ provides a measure for this purpose, which can be given by `get_spectrum_anharmonicity()`. The method considers the transition that is the closest to the qubit transition and subsequently calculates the quotient between these two transitions and returns abs(1-quotient). A cutoff has been implemented, such that the returned anharmonicity is within $[0, 1]$.

[10]:
```
circuit.get_spectrum_anharmonicity()
```

[10]:
```
1
```

1.3.8 $T1$ time

The robustness against decoherence is a crucial property of a circuit. For this purpose, we developed three measures to evaluate the $T1$ time contribution due to quasiparticle tunneling,

[11]:
```
T1_qp = circuit.get_T1_quasiparticles()
print("Quasiparticles noise contribution T1 = {:e} s".format(T1_qp))
```

```
Quasiparticles noise contribution T1 = 3.223110e-04 s
```

dielectric loss,

[12]:
```
T1_c = circuit.get_T1_dielectric_loss()
print("Charge noise contribution T1 = {:e} s".format(T1_c))
```

```
Charge noise contribution T1 = 1.576770e-02 s
```

and inductive loss

[13]:
```
T1_f = circuit.get_T1_flux()
print("Flux noise contribution T1 = {:e} s".format(T1_f))
```

```
Flux noise contribution T1 = 2.246226e-02 s
```

This leads to an overall $T1$ time measure of:

[14]:
```
print("Total T1 = {:e} s".format( 1/( 1/T1_qp + 1/T1_c + 1/T1_f)))
```

```
Total T1 = 3.114747e-04 s
```

Please refer to our preprint for more details on how the noise contributions are calculated.

1.3.9 Parameter sweep

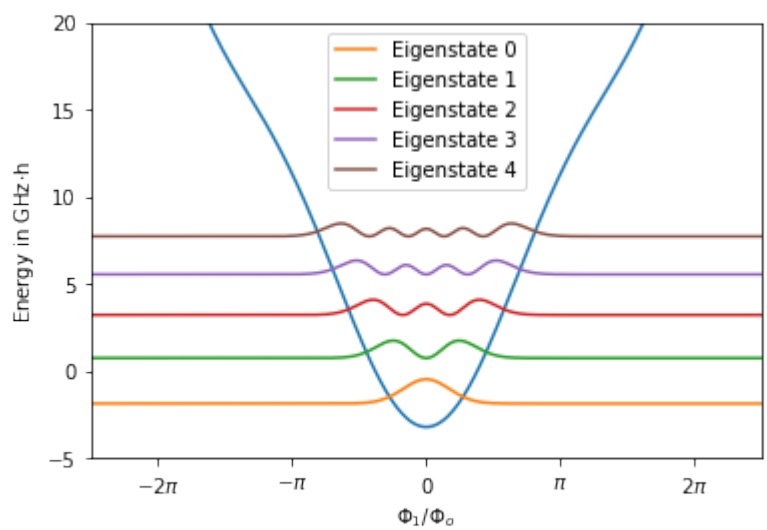
The dependence of circuit properties on the value of one or more parameters is a crucial aspect to consider when analysing superconducting circuits.

The exemplary spectrum above depends on the value of the external flux $\tilde{\Phi}_{010}$. For the previous plot, we have chosen the external flux to be $\pi \cdot \Phi_o$. If we set this value to 0, the spectrum has the following form:

[15]:
```
phi_ext = 0

circuit.get_numerical_hamiltonian(401, parameter_values=[False, EJ, False, phi_ext ])
eigv, eigs = circuit.get_eigensystem()

plt.plot(circuit.flux_list, np.array(circuit.potential)*y_scaling, lw=1.5)
for n in range(5):
    plt.plot(circuit.flux_list,
             (eigv[n]+ abs(eigs[:,n])**2*2e-23)*y_scaling
             ,label="Eigenstate " +str(n))
plt.legend()
plt.xlabel(r"$\Phi_1 / \Phi_o$")
plt.ylabel(r"Energy in GHz$\cdot$h")
plt.xticks(np.linspace(-2*np.pi, 2*np.pi, 5)*circuit.phi_0 ,
           [r'$-2\pi$', r'$-\pi$',r'$0$',r'$\pi$',r'$2\pi$'])
plt.xlim(-2.5*np.pi*circuit.phi_0, 2.5*np.pi*circuit.phi_0)
plt.ylim(-5,20)
plt.show()
```



The harmonicity of the spectrum increased due to the change of the external flux. This fact will also be reflected when calculating the anahrmonicity.

[16]:
```
circuit.get_spectrum_anharmonicity()
```

[16]:
```
0.05288147513281605
```

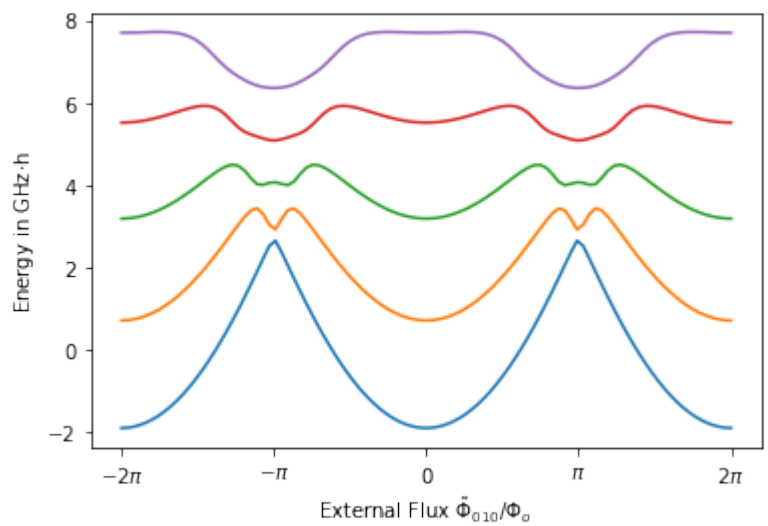
We can further study the dependence of the lowest eigenenergies as a function of the external flux.

Let's start by calculating the eigenvalues for differnt values of the external flux. Note that the order of the `paramter_values` list stays the same.

[17]:
```
eigenvalues = []
phi_ext_list = np.linspace(-2*np.pi*circuit.phi_0,2*np.pi*circuit.phi_0,100)
for phi_ext in phi_ext_list:
    circuit.get_numerical_hamiltonian(401, parameter_values=[False, EJ, False, phi_ext])
    eigv, eigs = circuit.get_eigensystem(5)
    eigenvalues.append(eigv)
```

Finally, we can plot the parameter sweep.

[18]:
```
plt.plot(phi_ext_list, np.array(eigenvalues)*y_scaling)
plt.xlabel(r"External Flux $ \tilde{\Phi}_{010} / \Phi_o$")
plt.ylabel(r"Energy in GHz$\cdot$h")
plt.xticks(np.linspace(-2*np.pi, 2*np.pi, 5)*circuit.phi_0 ,
           [r'$-2\pi$', r'$-\pi$',r'$0$',r'$\pi$',r'$2\pi$'])
plt.show()
```



We are not restricted in analyzing only the impact of the external flux, as this variation can be done for any parameter in `paramter_values`.

1.3.10 Charge offset

The offset of a node charge can be tuned by an external source. To work with charge offsets, the corresponding nodes have to be specified when initializing a `CircuitQ` instance. As a demonstration let's choose node 1 to have a charge offset.

[19]:
```
circuit = cq.CircuitQ(graph, offset_nodes=[1])
circuit.h
```

[19]:

$$-E_{J010}\cos\left(\frac{\Phi_1}{\Phi_o}\right)+\frac{\left(\Phi_1+\tilde{\Phi}_{010}\right)^2}{2L_{010}}+\frac{0.5\left(\tilde{q}_1+q_1\right)^2}{C_{01}}$$

The node charge q_1 is now shifted by an offset charge $\tilde{q}_1$. This new variable now also appears in in the parameter list:

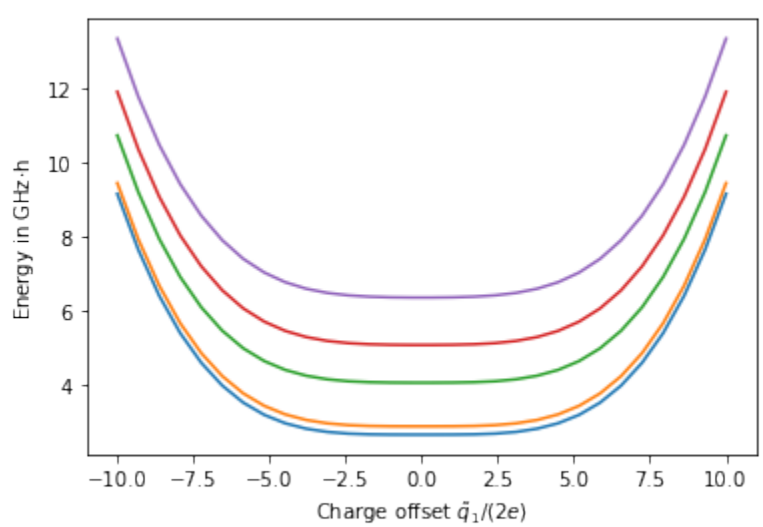
[20]:
```
circuit.h_parameters
```

[20]:
```
[C_{01}, E_{J010}, L_{010}, \tilde{\Phi}_{010}, \tilde{q}_{1}]
```

Finally, we can also study the impact of the external charge on the spectrum of the exemplary circuit.

[21]:
```
eigenvalues = []
phi_ext = np.pi*circuit.phi_0
q_off_list = np.linspace(-10*2*circuit.e,10*2*circuit.e,30)
for q_off in q_off_list:
    circuit.get_numerical_hamiltonian(401,
                parameter_values=[False, EJ, False, phi_ext, q_off])
    eigv, eigs = circuit.get_eigensystem(5)
    eigenvalues.append(eigv)

plt.plot(q_off_list/(2*circuit.e), np.array(eigenvalues)*y_scaling)
plt.xlabel(r"Charge offset $\tilde{q}_1 / (2e)$")
plt.ylabel(r"Energy in GHz$\cdot$h")
plt.show()
```



1.4 More Examples

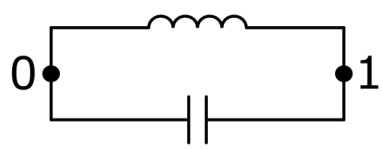
In this section you will find additional examples which supplement the exemplary circuits mentioned in the demonstration or tutorial section.

1.4.1 LC Circuit

The LC circuit is the most fundamental example. It behaves like a quantum harmonic oscillator in the quantum regime. We can test this fundamental property here.

[1]:
```
import circuitq as cq
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
```

Circuit graph



The circuit consists of a linear inductance with a shunted capacity.

[2]:
```
graph = nx.MultiGraph()
graph.add_edge(0,1, element = 'C')
graph.add_edge(0,1, element = 'L');
```

Symbolic Hamiltonian

The symbolic Hamiltonian contains a harmonic potential.

[3]:
```
circuit = cq.CircuitQ(graph)
circuit.h
```

[3]: $$\frac{\Phi_1^2}{2L_{010}} + \frac{0.5q_1^2}{C_{01}}$$

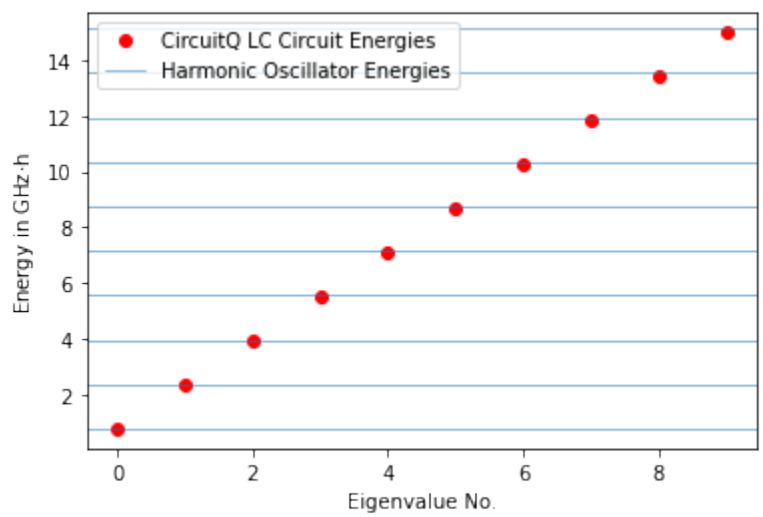
[3]: $$\frac{\Phi_1^2}{2L_{010}} + \frac{0.5q_1^2}{C_{01}}$$

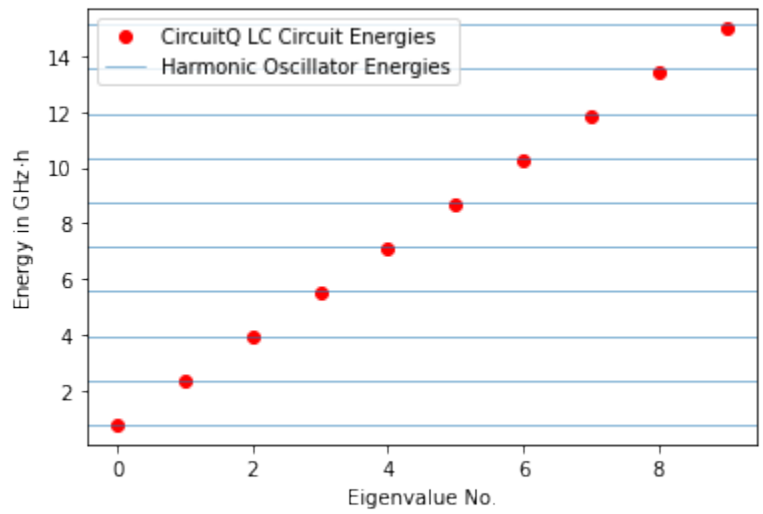
Diagonalization

[4]:
```
h_num = circuit.get_numerical_hamiltonian(200)
eigv, eigs = circuit.get_eigensystem()
```

As the LC circuit represents the circuit analog of a quantum harmonic oscillator, it's eigenenergies should be spaced by $\hbar\omega$ with the angular frequency $\omega = \frac{1}{\sqrt{LC}}$. We indicate these equidistant energies with horizontal lines below and plot the eigenvalues on top of them.

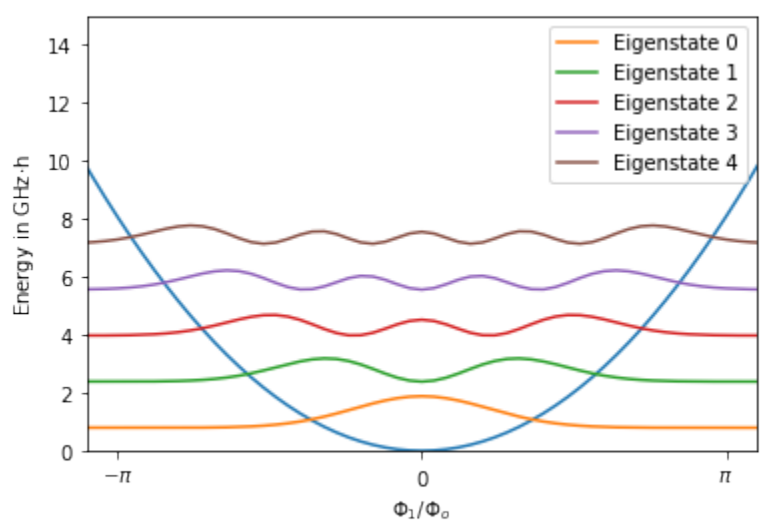
[5]:
```
n_energies = 10
h = 6.62607015e-34
y_scaling = 1/(h *1e9)
plt.plot(np.arange(n_energies), eigv[:n_energies]*y_scaling,
         'ro', label='CircuitQ LC Circuit Energies')
omega = 1/np.sqrt(circuit.c_v["L"]*circuit.c_v["C"])
plt.axhline(eigv[0]*y_scaling, lw=0.5, label='Harmonic Oscillator Energies')
for n in range(1,n_energies):
    plt.axhline((eigv[0]+n*circuit.hbar*omega)*y_scaling, lw=0.5)
plt.xlabel("Eigenvalue No.")
plt.ylabel(r"Energy in GHz$\cdot$h")
plt.legend()
plt.show()
```

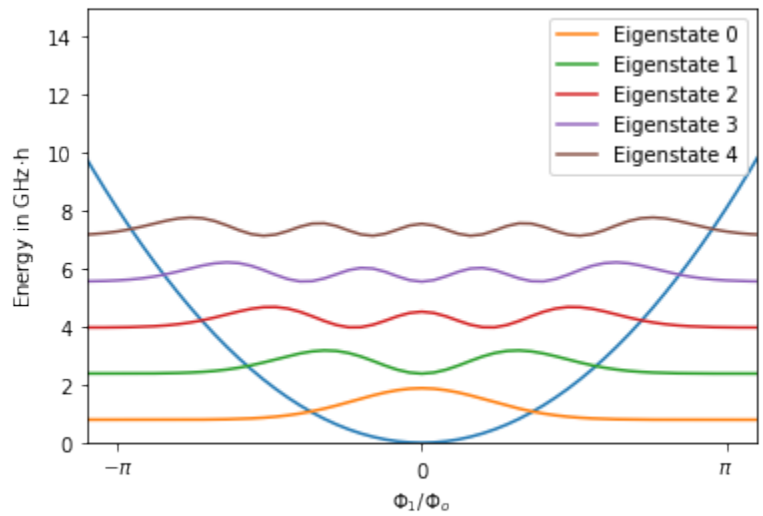




The eigenstates should have the shape of Hermite functions. Let's plot the square of their absolute value.

[6]:
```
plt.plot(circuit.flux_list, np.array(circuit.potential)*y_scaling, lw=1.5)
for n in range(5):
    plt.plot(circuit.flux_list,
             (eigv[n]+(abs(eigs[:,n])**2)*1e-23)*y_scaling,
             label="Eigenstate " + str(n))
plt.xticks(np.linspace(-1*np.pi, 1*np.pi, 3)*circuit.phi_0 ,
           [r'$-\pi$',r'$0$',r'$\pi$'])
plt.xlabel(r"$\Phi_1 / \Phi_o$")
plt.ylabel(r"Energy in GHz$\cdot$h")
plt.xlim(-1.1*np.pi*circuit.phi_0, 1.1*np.pi*circuit.phi_0)
plt.ylim(0,15)
plt.legend()
plt.show()
```



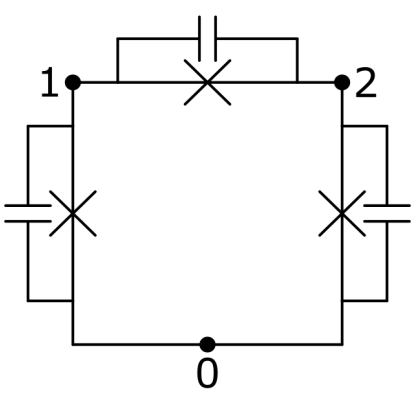


1.4.2 Flux Qubit

The flux qubit complements the previous circuits, which have been discussed in this documentation, as another important example since it represents a 3-node circuit. One of the nodes will be declared as a ground node which leads effectively to a 2-dimensional problem that will be discussed here.

[1]:
```
import circuitq as cq
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
```

Circuit Graph



[2]:
```
graph = nx.MultiGraph()
graph.add_edge(0,1, element = 'C')
graph.add_edge(0,1, element = 'J')
graph.add_edge(1,2, element = 'C')
graph.add_edge(1,2, element = 'J')
```

(continues on next page)

(continued from previous page)

```
graph.add_edge(0,2, element = 'C')
graph.add_edge(0,2, element = 'J');
```

Symbolic Hamiltonian

[3]:
```
circuit = cq.CircuitQ(graph)
circuit.h_parameters
```

[3]:
```
[C_{01}, C_{02}, C_{12}, E_{J010}, E_{J020}, E_{J120}, \tilde{\Phi}_{120}]
```

[4]:
```
circuit.h
```

[4]:

$$-E_{J010}\cos\left(\frac{\Phi_1}{\Phi_o}\right) - E_{J020}\cos\left(\frac{\Phi_2}{\Phi_o}\right) - E_{J120}\cos\left(\frac{-\Phi_1+\Phi_2+\tilde{\Phi}_{120}}{\Phi_o}\right) + 0.5q_1\left(\frac{C_{12}q_2}{C_{01}C_{02}+C_{01}C_{12}+C_{02}C_{12}} + \frac{q_1\left(C_{02}+C_{12}\right)}{C_{01}C_{02}+C_{01}C_{12}+C_{02}C_{12}}\right) + 0.5q_2\left(\frac{C_{12}q_1}{C_{01}C_{02}+C_{01}C_{12}+C_{02}C_{12}} + \frac{q_2\left(C_{01}+C_{12}\right)}{C_{01}C_{02}+C_{01}C_{12}+C_{02}C_{12}}\right)$$

(horizontally scroll the above LaTeX output)

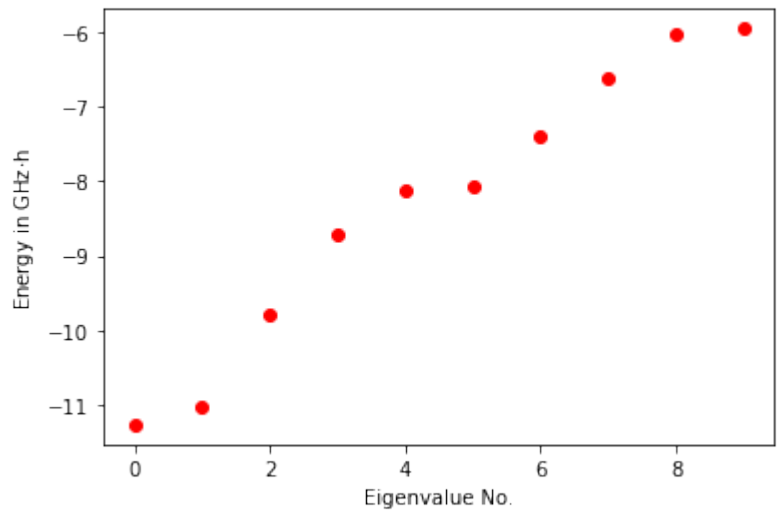
Diagonalization

Here, we will choose two junctions to have the same Josephson energy and the same shunted capacity. The third junction is scaled by `alpha`, which we choose to be 0.7. The parameter `dim` refers to the dimension for one node which leads to a total matrix dimension of `dim` $ ^2$.

[5]:
```
dim = 50
EJ = 1*circuit.c_v["E"]
alpha = 0.7
C = circuit.c_v["C"]
phi_ext = np.pi*circuit.phi_0
h_num = circuit.get_numerical_hamiltonian(dim,
                    parameter_values=[C,C,alpha*C,EJ,EJ,alpha*EJ,phi_ext])
eigv, eigs = circuit.get_eigensystem(100)
```

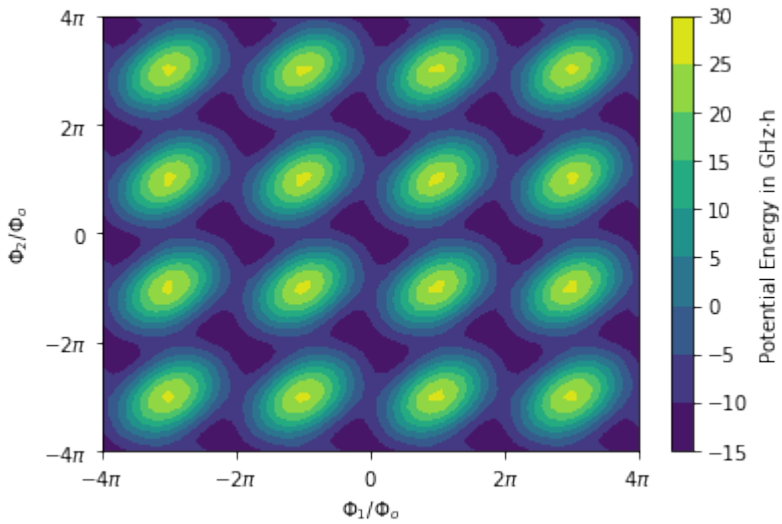
Let's plot the lowest eigenenergies.

[6]:
```
h = 6.62607015e-34
y_scaling = 1/(h *1e9)
plt.plot(np.arange(10), eigv[:10]*y_scaling, 'ro')
plt.xlabel("Eigenvalue No.")
plt.ylabel(r"Energy in GHz$\cdot$h")
plt.show()
```



The potential of the flux qubit is made up of three cosine terms. This leads to a egg carton shaped potential. As we didn't specify the grid length above, the flux coordinate grid is set to the default interval $[-4\pi\Phi_0, 4\pi\Phi_0]$. This will repeat the maximum 4 by 4 times. Let's plot the potential.

[7]:
```
def potential(phi_1, phi_2, phi_ex):
    return (-EJ*np.cos(phi_1/circuit.phi_0) - EJ*np.cos(phi_2/circuit.phi_0) -
              alpha*EJ*np.cos((phi_2-phi_1+phi_ex)/circuit.phi_0) )
phis = np.linspace(-4*np.pi*circuit.phi_0, 4*np.pi*circuit.phi_0, dim)
potential_list = [potential(phi_1,phi_2,phi_ext)*y_scaling
                  for phi_2 in phis for phi_1 in phis]
plt.contourf(phis, phis, np.array(potential_list).reshape(dim,dim))
plt.xticks(np.linspace(-4*np.pi, 4*np.pi, 5)*circuit.phi_0 ,
           [r'$-4\pi$',r'$-2\pi$',r'$0$',r'$2\pi$',r'$4\pi$'])
plt.yticks(np.linspace(-4*np.pi, 4*np.pi, 5)*circuit.phi_0 ,
           [r'$-4\pi$',r'$-2\pi$',r'$0$',r'$2\pi$',r'$4\pi$'])
plt.xlabel(r"$\Phi_1 / \Phi_o$")
plt.ylabel(r"$\Phi_2 / \Phi_o$")
plt.colorbar(label="Potential Energy in GHz$\cdot$h")
plt.show()
```

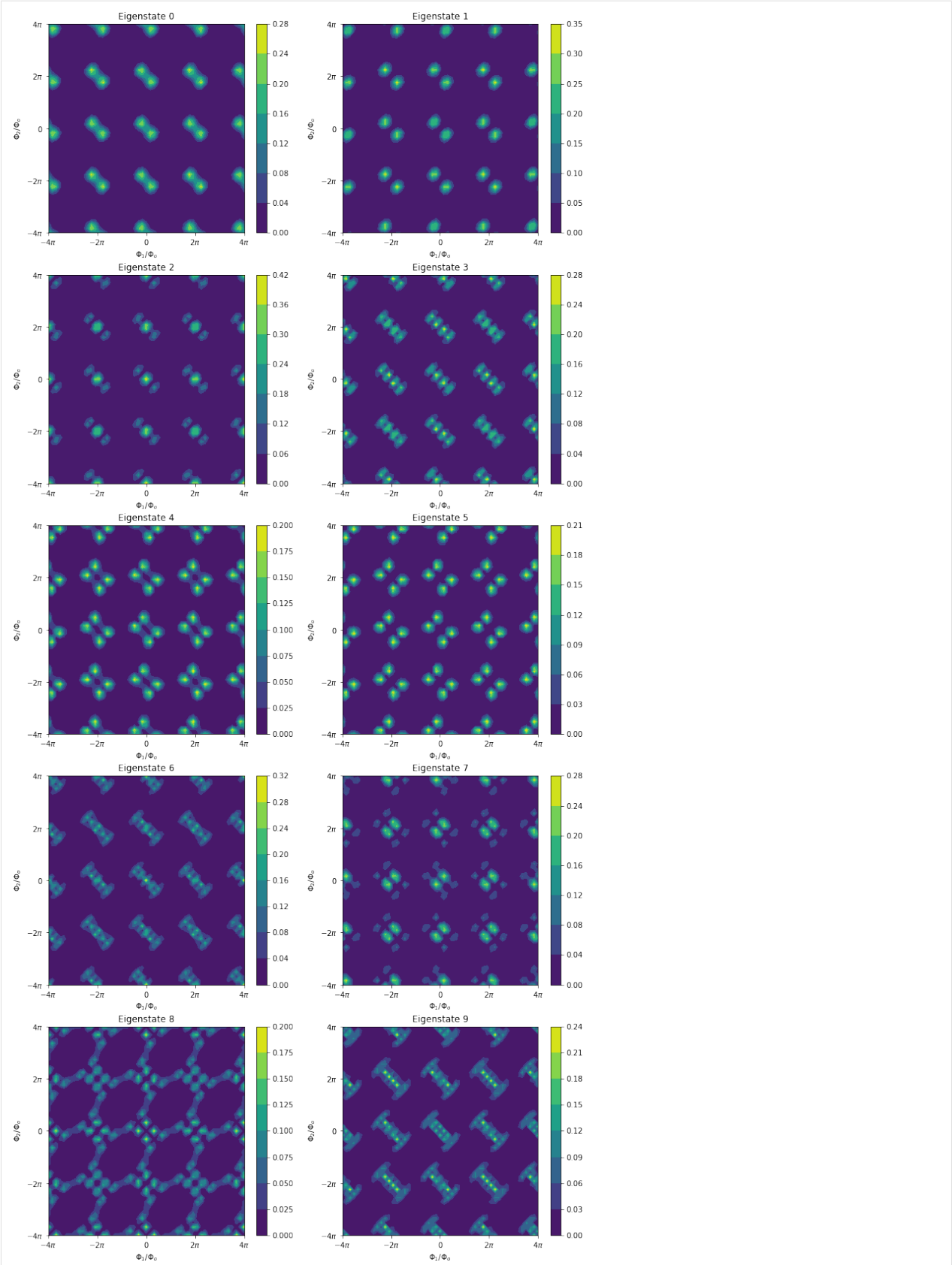


Due to the periodicity of the potential, CircuitQ implements the Hamiltonian in the charge basis. To be able to plot the eigenstates as a function of flux, we use the transformation method `transform_charge_to_flux()`.

[8]:
```
circuit.transform_charge_to_flux()
eigs = circuit.estates_in_phi_basis
```

Now, we are able to plot the square of the absolute value of the lowest eigenstates as a function of flux.

[9]:
```
plt.figure(figsize=(13,30))
for n in range(10):
    plt.subplot(5,2, n+1)
    plt.contourf(circuit.flux_list, circuit.flux_list,
      abs(np.array(eigs[n].reshape(circuit.n_dim,circuit.n_dim)).transpose())**2)
    plt.colorbar()
    plt.title("Eigenstate " + str(n) )
    plt.xticks(np.linspace(-4*np.pi, 4*np.pi, 5)*circuit.phi_0 ,
               [r'$-4\pi$',r'$-2\pi$',r'$0$',r'$2\pi$',r'$4\pi$'])
    plt.yticks(np.linspace(-4*np.pi, 4*np.pi, 5)*circuit.phi_0 ,
               [r'$-4\pi$',r'$-2\pi$',r'$0$',r'$2\pi$',r'$4\pi$'])
    plt.xlabel(r"$\Phi_1 / \Phi_o$")
    plt.ylabel(r"$\Phi_2 / \Phi_o$")
plt.show()
```


Eigenstate 0
Eigenstate 1
Eigenstate 2
Eigenstate 3
Eigenstate 4
Eigenstate 5
Eigenstate 6
Eigenstate 7
Eigenstate 8
Eigenstate 9
Φ_1/Φ_o
Φ_2/Φ_o

We find that the lowest two eigenstates are located at the double well between the maxima of the potential.

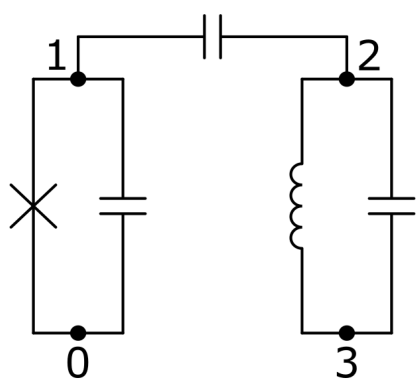
1.4.3 Transmon coupled to a resonator

This example is fundamentally different to the other circuits discussed in this documentation in two ways. First, it does not describe a single system or qubit, but the coupling of a qubit (here: transmon) with another system (here: resonator). With its current implementation design, CircuitQ treads the entire circuit as a single system and provides the description of the whole system with one Hamiltonian. Second, the numerical implementation of this circuit will be provided by a mixture of the charge and flux basis, as the copuling capacity seperates the resonator as a harmonic system implemented in the flux basis from the transmon, which is implemented in the charge basis.

[1]:
```
import circuitq as cq
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
```

Circuit Graph

We initialize a transmon coupled to a resonator (LC circuit) via a capacity.



[2]:
```
graph = nx.MultiGraph()
graph.add_edge(0,1, element = 'C')
graph.add_edge(0,1, element = 'J')
graph.add_edge(1,2, element = 'C')
graph.add_edge(3,2, element = 'C')
graph.add_edge(3,2, element = 'L');
```

Symbolic Hamiltonian

We will set node 0 and 3 to the ground.

[3]:
```
circuit = cq.CircuitQ(graph, ground_nodes=[0,3])
```

[4]:
```
circuit.h
```

[4]:
$$-E_{J010}\cos\left(\frac{\Phi_1}{\Phi_o}\right) + 0.5q_1\left(\frac{C_{12}q_2}{C_{01}C_{12}+C_{01}C_{23}+C_{12}C_{23}} + \frac{q_1\left(C_{12}+C_{23}\right)}{C_{01}C_{12}+C_{01}C_{23}+C_{12}C_{23}}\right) + 0.5q_2\left(\frac{C_{12}q_1}{C_{01}C_{12}+C_{01}C_{23}+C_{12}C_{23}} + \frac{q_2\left(C_{01}+C_{12}\right)}{C_{01}C_{12}+C_{01}C_{23}+C_{12}C_{23}}\right) + \frac{\Phi_2^2}{2L_{230}}$$

(horizontally scroll the above LaTeX output)

The Hamiltonian contains a periodic cosine potential caused by the transmon an the harmonic part of the resonator. The nodes associated with the transmon will be implemented in the charge basis, which is also reflected in the value of `charge_basis_nodes`:

[5]:
```
circuit.charge_basis_nodes
```

[5]:
```
[0, 1]
```

To prevent the usage of the charge basis, you can specify the list of nodes for which a flux basis implementation should be forced with the keyword `force_flux_nodes` when initialising a `CircuitQ` instance.

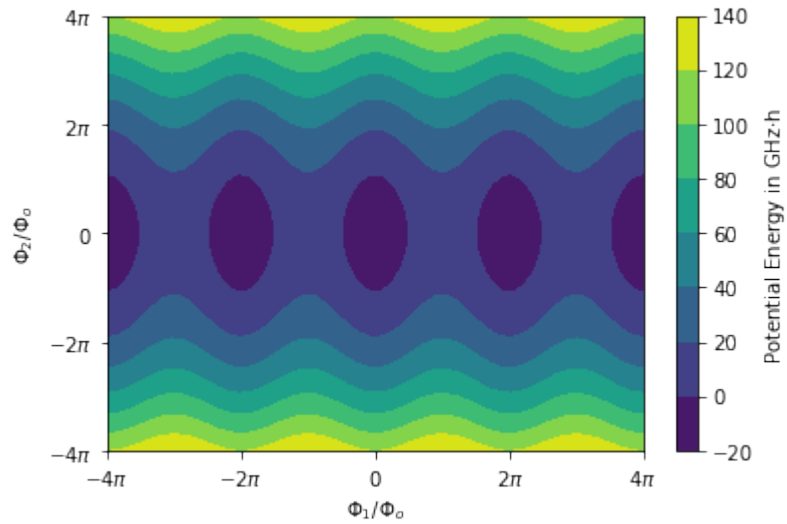
Diagonalization

The calculation of eigenstates and eigenvalues of the Hamiltonian is done in the mixed basis. To be able to plot the eigenstates in the flux basis, we use the method `transform_charge_to_flux()` and subsequently address the eigensates stored in `estates_in_phi_basis`.

[6]:
```
dim = 50
h_num = circuit.get_numerical_hamiltonian(50)
eigv, eigs = circuit.get_eigensystem()
circuit.transform_charge_to_flux()
eigs = circuit.estates_in_phi_basis
```

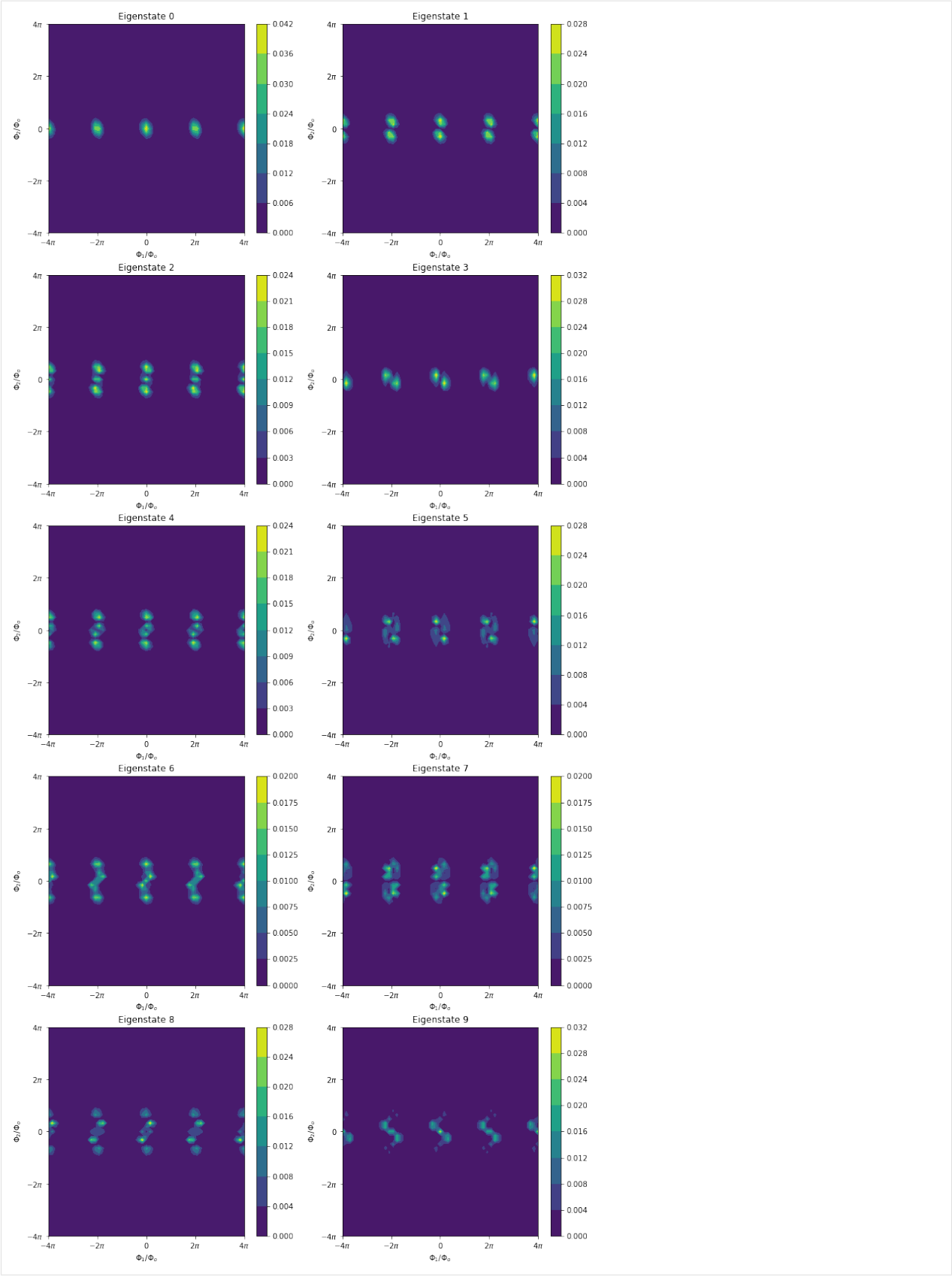
Let's plot the potential, which is an overlay of the cosine potential in one direction and a parabola in the other direction.

[7]:
```
def potential(phi_1, phi_2):
    return (-circuit.c_v['E']*np.cos(phi_1/circuit.phi_0) + phi_2**2/(2*circuit.c_v['L
↪']))
phis = np.linspace(-4*np.pi*circuit.phi_0, 4*np.pi*circuit.phi_0, dim)
h = 6.62607015e-34
y_scaling = 1/(h *1e9)
potential_list = [potential(phi_1,phi_2)*y_scaling for phi_2 in phis for phi_1 in phis]
plt.contourf(phis, phis, np.array(potential_list).reshape(dim,dim))
plt.xticks(np.linspace(-4*np.pi, 4*np.pi, 5)*circuit.phi_0 ,
           [r'$-4\pi$',r'$-2\pi$',r'$0$',r'$2\pi$',r'$4\pi$'])
plt.yticks(np.linspace(-4*np.pi, 4*np.pi, 5)*circuit.phi_0 ,
           [r'$-4\pi$',r'$-2\pi$',r'$0$',r'$2\pi$',r'$4\pi$'])
plt.xlabel(r"$\Phi_1 / \Phi_o$")
plt.ylabel(r"$\Phi_2 / \Phi_o$")
plt.colorbar(label="Potential Energy in GHz$\cdot$h")
plt.show()
```



Finally, we plot the lowest eigenstates depicted in the flux basis.

[8]:
```
plt.figure(figsize=(13,30))
for n in range(10):
    plt.subplot(5,2, n+1)
    plt.contourf(circuit.flux_list, circuit.flux_list,
        abs(np.array(eigs[n].reshape(circuit.n_dim,circuit.n_dim)).transpose())**2)
    plt.colorbar()
    plt.title("Eigenstate " + str(n) )
    plt.xticks(np.linspace(-4*np.pi, 4*np.pi, 5)*circuit.phi_0 ,
               [r'$-4\pi$',r'$-2\pi$',r'$0$',r'$2\pi$',r'$4\pi$'])
    plt.yticks(np.linspace(-4*np.pi, 4*np.pi, 5)*circuit.phi_0 ,
               [r'$-4\pi$',r'$-2\pi$',r'$0$',r'$2\pi$',r'$4\pi$'])
    plt.xlabel(r"$\Phi_1 / \Phi_o$")
    plt.ylabel(r"$\Phi_2 / \Phi_o$")
plt.show()
```


Eigenstate 0
Eigenstate 1
Eigenstate 2
Eigenstate 3
Eigenstate 4
Eigenstate 5
Eigenstate 6
Eigenstate 7
Eigenstate 8
Eigenstate 9
Φ_1/Φ_0
Φ_2/Φ_0
-4π
-2π
0
2π
4π

1.5 API reference

1.5.1 CircuitQ class

class circuitq.core.**CircuitQ**(*circuit_graph*, *ground_nodes=None*, *offset_nodes=None*, *force_flux_nodes=None*, *print_feedback=False*, *natural_units=False*)

Bases: `object`

A class corresponding to a superconducting circuit.

get_T1_dielectric_loss(*excited_level=None*)

Estimates the T1 contribution due to dielectric loss. See the preprint on arXiv for more details about the formulas that have been used for this method.

Parameters

excited_level (*int (Default: Number of first state above the groundstate)*) – Number of state, which is considered to be the excited state.

Returns

T1_dielectric_loss – T1 contribution due to dielectric loss.

Return type

float

get_T1_flux(*excited_level=None*, *lower_bound=False*)

Estimates the T1 contribution due to flux noise. See the preprint on arXiv for more details about the formulas that have been used for this method.

Parameters

- **excited_level** (*int (Default: Number of first state above the groundstate)*) – Number of state, which is considered to be the excited state.
- **lower_bound** (*bool (Default:False)*) – If set to True, a lower bound for the T1 contribution will be calculated by summing over all inductive branches (not just closure branches as in default mode)

Returns

T1_flux – T1 contribution due to flux noise.

Return type

float

get_T1_quasiparticles(*excited_level=None*)

Estimates the T1 contribution due to quasiparticles. See the preprint on arXiv for more details about the formulas that have been used for this method.

Parameters

excited_level (*int (Default: Number of first state above the groundstate)*) – Number of state, which is considered to be the excited state.

Returns

T1_quasiparticle – T1 contribution due to quasiparticles.

Return type

float

get_classical_hamiltonian()

Returns a Hamiltonian as a Sympy-function for the circuit.

Parameters
parameters. (*No external*) –

Returns

- **h** (*Sympy Add*) – Symbolic Hamiltonian of the circuit
- **h_parameters** (*list*) – List of parameters in the Hamiltonian

get_eigensystem(*n_eig=30*)

Calculates eigenvectors and eigenstates of the numerical Hamiltonian.

Parameters
n_eig (*int*) – Number of returned eigenvalues.

Returns

- **evals** (*array*) – Array of eigenvalues
- **estates** (*array*) – Array of eigenstates

get_numerical_hamiltonian(*n_dim*, *grid_length=None*, *unit_cell=False*, *parameter_values=None*, *default_zero=True*)

Creates a numerical representation of the Hamiltonian using the finite difference method.

Parameters

- **n_dim** (*int*) – Dimension of numerical matrix of each subsystem. If an even number is given, the value is set to the next odd number. This is done to match the dimension of matrices between the charge and flux basis.
- **grid_length** (*float (Default 4*np.pi*phi_0)*) – The coordinate grid is taken from -grid_length to +grid_length in n_dim steps.
- **unit_cell** (*bool (Default False)*) – If set to True, the coordinate grid for the flux variables is taken to capture the unit cell of the potential. If False, grid_length is used to determine the coordinate grid.
- **parameter_values** (*list*) – Numerical values of system parameters (corresponds to self.h_parameters).
- **default_zero** (*bool (Default True)*) – If set to True, the default valeus of the charge and flux offsets will be 0.

Returns
h_num – Numeric Hamiltonian of the circuit given as a matrix in array form

Return type
numpy array

get_spectrum_anharmonicity(*nbr_check_levels=3*)

Calculates the anharmonicity of the eigenspectrum. The method can handle degenerated eigenenergies. It considers the transition that is the closest to the qubit transition (groundstate-first excited states) and subsequently calculates the quotient between these two transitions and returns abs(1-quotient).

Parameters
nbr_check_levels (*int (Default 3)*) – Number of levels that should be considered for this analysis. The counting includes groundstate and first excited state.

Returns
anharmonicity – abs(1-quotient) as described above

Return type
float

transform_charge_to_flux()

Transforms the eigenvectors into the flux basis. This is necessary to plot the states as a function of flux if the numerical Hamiltonian was (partially) implemented in the charge basis.

Parameters

parameters (*No external*) –

Returns

estates_in_phi_basis – An array that contains the eigenstates in the flux basis.

Return type

array

1.5.2 circuitq.functions_file module

circuitq.functions_file.**visualize_circuit_general**(*graph*, *save_as*)

Visualises a circuit by creating a figure to an arbitrary path.

Parameters

- **graph** (*NetworkX Graph*) – Input circuit
- **save_as** (*String*) – Arbitrary figure path

Note: Please refer to our preprint on arXiv for more details on physics and implementation and to cite our project.

The source code can be found on CircuitQ's repository at GitHub.

CircuitQ was developed by Philipp Aumann and Tim Menke under the supervision of William Oliver and Wolfgang Lechner.

CHAPTER

TWO

INDEX

PYTHON MODULE INDEX

C

INDEX

C

G

M

T

V