Qubit Simulation Module

Colorado School of Mines

Developer: Joel Howard, joelhoward@mines.edu

# Introduction

The Qubit Simulation Module (QSM) is a software package that facilitates the full design process for superconducting qubits and resonators. It is designed to be highly modular and accommodate any user-defined models, but by default it uses those found in Chapter 4 of Junling Long’s doctoral thesis found [here](https://sites.google.com/view/junlings-research-homepage/thesis?authuser=0). It is written in Python with external calls to the Ansys HFSS, Q3D, and Nexxim simulators.

# For Users

## Setup

To use the QSM, the user must have file access to:

* …the Ansys executable, usually somewhere like ProgramFiles/AnsysEM/AnsysEM19.5/Win64/ansysedt.exe”
* …the QSMSource folder.

The python environment must also have the following installed:

* Sympy
* gdspy
* shapely
* numpy
* matplotlib
* pathlib

To begin a set of simulations, the user must have a fresh directory containing a template version of “qubitSimulator.py”. To permit it to import qubitSimulationModule.py, it should have a header that looks something like:

QSMSourceFolder=Path("O:/68707/JoelHoward/ChipDesign/QSMSource/")

sys.path.append(QSMSourceFolder)

import qubitSimulationModule as QSM

computeLocation="Windows"

projectFolder=Path(os.path.dirname(os.path.abspath( \_\_file\_\_ )))

## Workflow

The fundamental simulation unit of the QDSM is a folder with an instance of qubitSimulator.py. It is one single device design – this folder will not contain parameter sweeps or sub-designs. These can be coordinated by a user-written automation script, which will not be discussed in this documentation. All discussion hereafter refers to “design” as a single-folder unit.

The design/simulation parameters are stored in various CSV files in the folder, which are programmatically generated by qubitSimulator.py. There are three central design files:

* [name]\_simulationParametersFile.csv
* [name]\_componentParametersFile.csv
* [name]\_componentGeometriesFile.csv

The simulation files are stored in their respective sub-folders.

The user’s simulation workflow consists of editing qubitSimulator.py such that it will perform the desired task, then running it. For a fresh simulation this workflow would begin with

QSM.generateSystemParametersFile(projectFolder)

to initialize the system parameters file. Then, all subsequent runs of qubitSimulator **must begin with**

qSys=QSM.initialize(projectFolder,computeLocation,QSMSourceFolder)

qSys is a python object that stores *all the information about the qubit system*. All data loaded from the layout and simulation files is stored in it, so it can be a powerful debugging/prototyping tool.

Next, the design files need to be generated and populated. This is achieved by running

qSys.generateFile(“componentParams”)

qSys.generateFile(“geometries”)

populating the generated CSV after each. Once the geometries file is generated the user can view the chip layout via

qSys.generateFile(“GDS”)

As soon as the geometries file is populated, simulations can be run. This is achieved via

qSys.simulationCommand(~)

where the ~ can be replaced by numerous commands. To start with, the user will typically run

* [“simulation”, “capMat”,”init”] to produce the parameters csv for a Q3D capacitance matrix simulation
* [“simulation”, “capMat”,”run”] to perform the Q3D capacitance matrix simulation. Note that “run” is reserved for doing something that involves either Ansys or a cluster.
* [“simulation”, “capMat”,”postProcess”] to post-process the Q3D capacitance matrix simulation.

All simulations have a similar init/run/postProcess sequence (though many skip “run”). In condensed notation moving forward this will be written e.g.

[“simulation”, “capMat”,{”init”,”run”,”postProcess”}]

After the capacitance matrix is post-processed, the user will likely want to run circuit simulations.

List of circuit simulations:

* General: fullS21 (i.e. [“simulation”, “fullS21”,{”init”,”run”,”postProcess”}])
* Qubit: freqQ[N], decayQ[N], exchQ[N]
* Resonator: freqR[N], lumpedR[N], feedlineCouplingR[N]

Once all circuit simulations are completed, the user can run:

* [“simulation”, “capMatGE”,{“init”,”postProcess”}] to calculate the gaussian-eliminated capMat simulation.
* [“simulation”, “ECQ[N]”,{”init”,”postProcess”}] to calculate the Ec of Q[N]
* [“simulation”, “ECR[N]”,{”init”,”postProcess”}] to calculate the Ec of R[N]
* ["simulation","L\_iQ[N]",{"init","postProcess"}] to calculate the L\_i, which we want to get as close to the L\_I specified in componentParameters.

Next the circuit can be quantized via

[“simulation”, “quantize”, {“init”,”postProcess”}]

(for cluster-based computing there is a “run” option that submits the quantization to be computed on a node). Note that in the simulation parameters file “QuantizeList” must be in the format i.e. “[Q0:R0:Q1]” (without the quotations) for a quantization that only considers Q0,R0, and Q1.

After quantization the post processing simulations currently available are :

* ["simulation","zzQ[m]-[n]",{"init","postProcess"}]# i.e. zzQ0-1 to calculate the zz coupling
* ["simulation","anharmonicityQ[N]",{"init","postProcess"}] to calculate the anharmonicities
* ["simulation","dispersiveShiftR[N]",{"init","postProcess"}] to calculate the dispersive shift of the readout resonator.

A list of helper commands that condense many of these steps after capMat has been run and post-processed are available:

* command=“initAllSims” to generate the initialization files for all circuit simulations. Set frequency sweeps.
* command=“runAllSims” to run all circuit simulations.
* command=“postProcessAllSims” to post-process all circuit simulations.
* command=“GEPlusAllEC” to calculate the gaussian eliminated capMat and all qubit Ec values.
* command=“Allzz” to calculate all ZZ couplings
* command=“AllAnharmonicityQ” to calculate all anharmonicities
* command=“AllDispersiveShiftR” to calculate all anharmonicities

## Options

Some fields in the input CSV files functionally act as a drop-down menu, and will only work for a specific set of options. Black=required, orange=optional.

* systemParametersFile
  + Material
    - perfect conductor, aluminum, copper -> see Ansys material list.
  + Flip Chip? Simulate Feedline?,Add Mesh to GDS?, Invert GDS?
    - Yes, No
  + Chip Markers
    - Pappas, Schmidt
  + Simulation
    - 2D,3D
* componentParametersFile
  + Qubit type
    - Required:{Floating,Grounded}-rectangularPads-{single,double}JJ-Finger\_Pad{1,2}\_Num[N]
      * Example: Floating-rectangularPads-doubleJJ-Fingers\_Pad1\_N3-Fingers\_Pad2\_N2
  + Resonator Pad[N] Type
    - T shaped pad
  + Control line type
    - “fluxBias”: grounded flux bias, launch pad at only one end.
    - “feedline”: Launch pad at both ends.
    - “drive”: Launch pad at just one end.
* componentGeometriesFile: Most of these definitions can be found in the “GeometryParameters” powerpoint. A few extra notes:
  + JJ Location must be specified as such: “[x:y]” (without the quotations). Here x is the shift to the right and y is the shift up of the normalized JJ location.

## Model Assumptions

* Qubits
  + Labelled “Q[N]”, where the index corresponds to the position of its resonator feedline pad along the feedline
* Substrate
  + Centered on (0,0)
  + For flip chip both chips are centered on (0,0), but can be rectangles of different sizes.
* Resonators
  + The pads are parallel rectangles. The edge closer to the other rectangle is where the CPW connects to.
  + **The turn radius must be greater than twice the mesh boundary parameter or the meander’s meshPeripheryPolyLine will not be a well-defined polygon.**
  + **The resonators must be numbered according to their position along the feedline, with 0 being the left-most (for a straight unrotated feedline). This is so the simulation knows how to model the feedline as a series of transmission lines of various lengths.**
* Control Lines
  + Start Angle is in radians
  + Section Code: This is how the path of the meander is specified. It is best explained by example. Consider (S:100)(R:-1.5708:100)(S:1900)(R:3.14159:100) This corresponds to a straight segment (“S”) of length 100um, followed by a turn (“R”) of angle -pi/2 radians and turn radius of 100um followed by a straight segment of length 1900um followed by a turn of angle pi and turn radius 100um. Any number of these parentheses-delimited S or R “codes” can be strung together to generate an arbitrary meander.
* Ground
  + Covers the full substrate
  + Separated from nodes via a spacing given by the node’s “boundary” parameters
* Control Lines
  + **There can be at most one feedline, and it must be designated as CL0.**
* Geometry coordinates
  + Implied view is top-down on the x-y plane, looking along the negative z-axis. Geometry references to “width” correspond to the x-dimension, “length” correspond to the y-dimension, and height/thickness correspond to the z-dimension, *except* for resonators and PTCs, in which “Length” corresponds to the actual length of the CPW (for resonators this includes the extra length due to the end pads).
* Model
  + Capacitances are, by and large, not generally removed even for distant components. That said, some models, such as the lumping of the resonator, rely on some capacitances to be negligible. Typically we want to operate in the regime where the following capacitances are negligible:
    - CR[N]Pad1\_R[N]Pad2
    - CR[N]Pad1\_R[M]Pad1,2, N!=M
    - CR[N]Pad2\_R[M]Pad1,2, N!=M

# For Developers