SpecialResonatorCalcs Documentation

Release 0.01

Gareth Sion Jones

Mar 11, 2020

Contents

Often when using coplanar waveguide resonators for quantum spin dynamics, it is desirable to understand certain critical figures of merit. However, the workflow required for determining these data is often quite laborious. This package details an automated procedure calling ssh protocols into remote machines..

Please note, this package is still as yet a work in progress. I will update a working 'ToDo' list frequently, and I will ocasionally be tidying up the code and folder sturcture.

CHAPTER 1

Guide

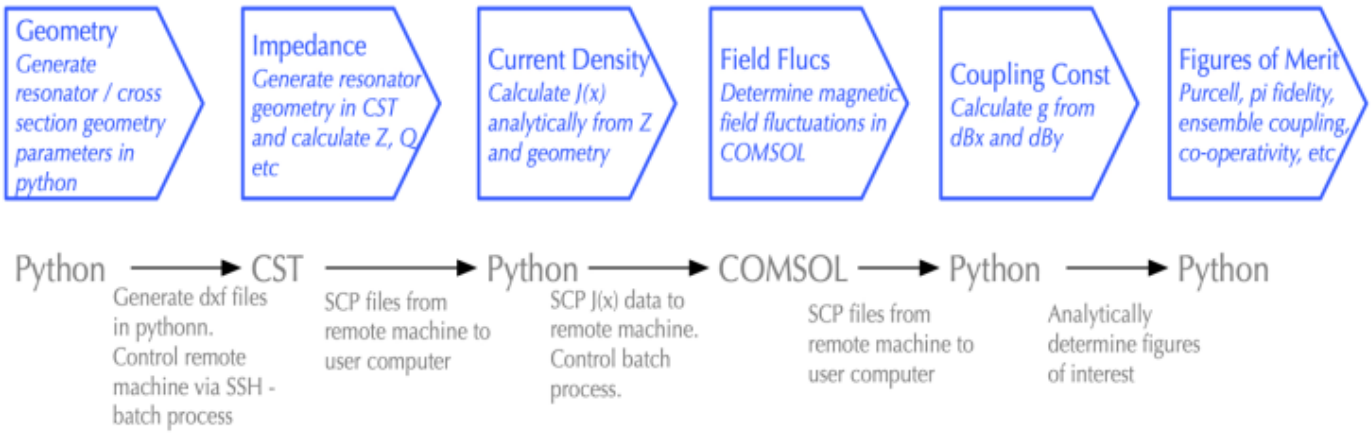
1.1 Overview

When using coplanar waveguide resonators for quantum spin dynamics, there is often a need to analytically/numerically determine specific figures of merit particular to the resonator design. The aim here is that we can assess how the design will behave for a given spin system and desired experiment. The workflow for performing these calculations is typically quite laborious however, and often involves using multiple software packages and handling great amounts of output data. It is desirable then to automate this process, such that a user specifies only very simple parameters, but receives the full range of numerical outputs.

This documentation details a package which achieves just that. Specifically, a user will input very simple geometric parameters for their CPW, such as conductor width and length, the gap between the conductor and ground, the conductor thickness, and the substrate dimensions. The impedance of this structure is then determined by runnning CST Microwave Studio from the command line via the ssh protocol. The data from cst is loaded into a preprocessing python script, where the vacuum current fluctations for the cpw are determined analytically.

This current density J(x) is then passed as a csv file to a remote machine, along with the geometric parameters of the resonator, via ssh. A pre-generated COMSOL script is updated with the new parameters, and then run via the command line remotely. Magnetic field data generated by COMSOL is autonomously transferred back to the user computer, where post-processing of the data can commence.

This workflow is summarized in the below figure



1.1.1 Package Overview

This is a multi-layered software package, which contains three library hierarchies:

1. electromagnetics
2. ssh_command
3. data_processing

1.2 Installation

This package has been designed for easy interfacing with Linux or Mac OS operating systems. It is possible to use this package on Windows operating systems, but the ssh commands may not work so well. I will modify the package for Windows at a later date.

This package has several dependencies. First, you must install numpy adnd scipy. The easiest way to do this is to use pip. If you do not have pip installed onn your compiter, it can be downloaded from the internet. Then, in a terminal winndow, type:

```
pip install matplotlib
pip install numpy
pip install scipy
```

To install QSDCalcs, open type the command:

```
pip install qsd
```

This will download the package to your computer. Next, you will need to download the COMSOL .mph file. This file is hosted at https://github.com/garethsion/qsd/blob/master/qsd/tests/cpw_vacuum_calcs.mph . Download this to your working directory.

1.3 Using the Package

There is a specific workflow which must be adhered to in order to use the package.

1. First, you must specify the cpw geometry parameters. I have specified a text file called cpw_parameters.txt where I update the values, and in a preprocessing script I simply read the parameters in from that file. The text file which specifies the parameters looks something like

```
w = 10e-06
t = 50e-09
l = 8.194e-03
pen = 200e-09
omega = 7.3e09
Z = 50
```

2. Next, you must define a preprocessing script which sets the cpw geometry parameters, and determines the current density of the structure based on those parameters. See the example current_density.py to get a better idea on how to achieve this.

3. With the current density calculated and stored in a parameter file, you can now remoteley connect to a host computer and run COMSOL. One way to achieve this is to define a python script such as the example remote_interface.py, and run that script from the terminal window with the command python remote_interface.py.

   There are a number of host computers available at ucl. Two common ones are vienna and monaco, found in the EEE depa
   ssh <USERNAME>@ee.ucl.ac.uk

   There is also gade.phys. However, this is my personal machine, and is often in use.

   This would remoteley give you access to the machine from your own computer. This package does all of this for you. When you first use the package, you will need to specify your username annd the host network (ee.ucl.ac.uk), and the ssh_command library includes functionality to gennerate a secure ssh key and upload this to the host computer.

   Bear in mind, if you want to ssh into a ucl machine when you are outside UCL, you will be blocked by the firewall. To get round this, you need to install the ucl vpn. This can be found on the UCL Software Database. When outside UCL, first login to the vpn, and then procedd as normal.

4. You may find it useful to add a remote machine to your ssh config file. This makes life a lot easier when transferring the files and interfacing with a remote machine. An example of how to do this is given in the example "Adding a remote machine"

5. The remote_interface script will copy the parameter list to the remote machine, update a predefined COMSOL script with the new parameters, remotely run COMSOL, and retrieve the data back to your computer.

6. With the data now retrieved, you can post-process the data to determine certain figures of merit. See the post-processing examples.

1.4 How the COMSOL Simulation Works

In order to model the vacuum field fluctuations in a superconducting microwave resonator, we model the structure as a two-dimenisional wire. This is a simple magnetic fields AC/DC simulation, and we are interested in solving Ampere's Law for the cpw structure.

The first step in the procedure is to calculate the spatial dependence of the vacuum current fluctuations. There are several different ways to do this, all of which are untilately equivalent. In this package at the moment I use two different methods, as i am testing which one is the easiest to deal with in simulation.

One method of finding the current density is given in the paper "Reaching the quantum limit of sensitivity in electron

spin resonance" (Bienfait, et al, 2015):

$$\delta J(x) = \begin{cases} \delta J(0)\left[1-(2x/w)^2\right]^{-1/2}, for\,|x| \leq \left|\frac{1}{2}w - \lambda^2/(2b)\right| \\ \delta Jt(\frac{1}{2}w) exp\left[(\frac{1}{2}w - |x|)b/\lambda^2\right], for\left|\frac{1}{2}w - \lambda^2/(2b)\right| \leq |x| \leq \frac{1}{2}w \\ (1.165/\lambda)(wb)^{1/2}\delta J(0), forx = \frac{1}{2}w \end{cases}$$

1.5 To Do List

This code is still a work in progress. There are quite a few things I still have to do. A detailed list is given below:

- Include EPR spectra code to calculate the eigenvalues of the EPR transitions. This then forms a key value going into the calculation for the single spin coupling, i.e. we need to calculate $\langle m_f | \hat{S}_x | m_f \rangle$ for the single spin coupling.

$$g = \langle m_f | \hat{S}_x | m_f \rangle \gamma_e \sqrt{\delta B_y^2 + (\cos\theta)\,\delta B_x^2}$$

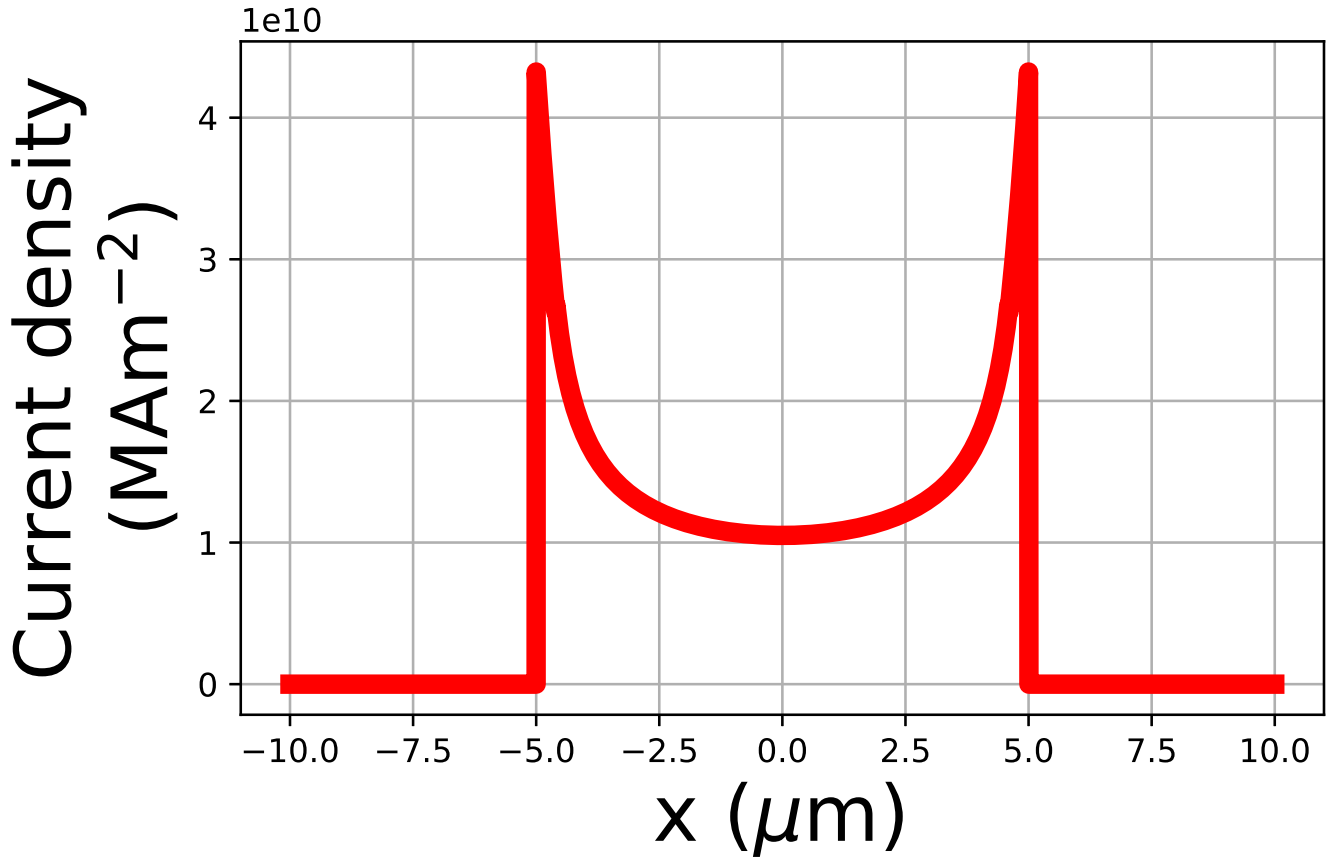
- Automate CST procedure to calculate impedance and external q factor
- Specify the distribution of spins for the single_spin_coupling example
- Weight the purcell_enhancement and pi pulse fidelity examples by contribution to the signal
- Apply a fidelity measure to the pi pulse

1.6 Gallery

Here are some figures generated using this package

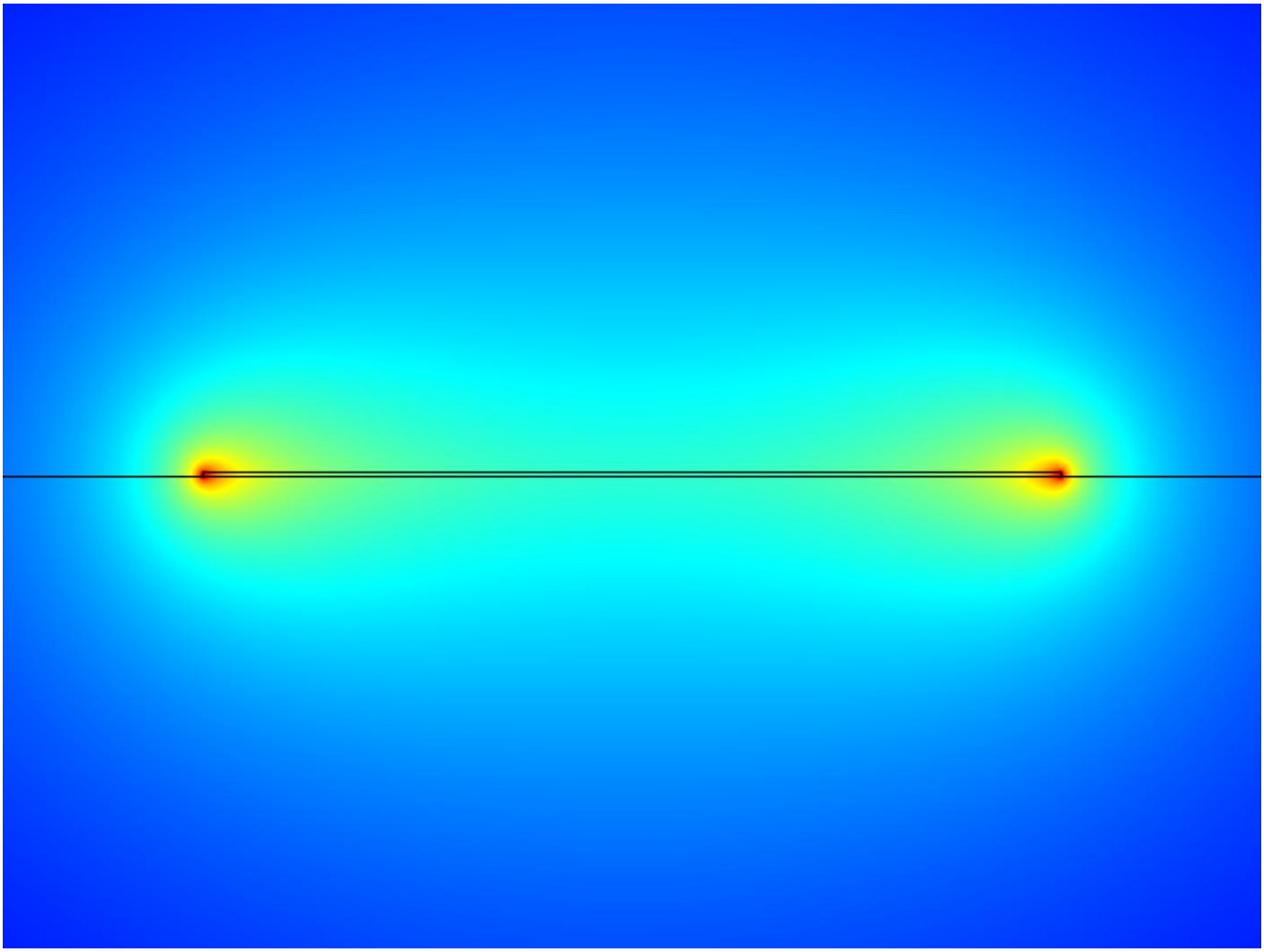
1.6.1 Electromagnetic Properties

1.) CPW vacuum current density

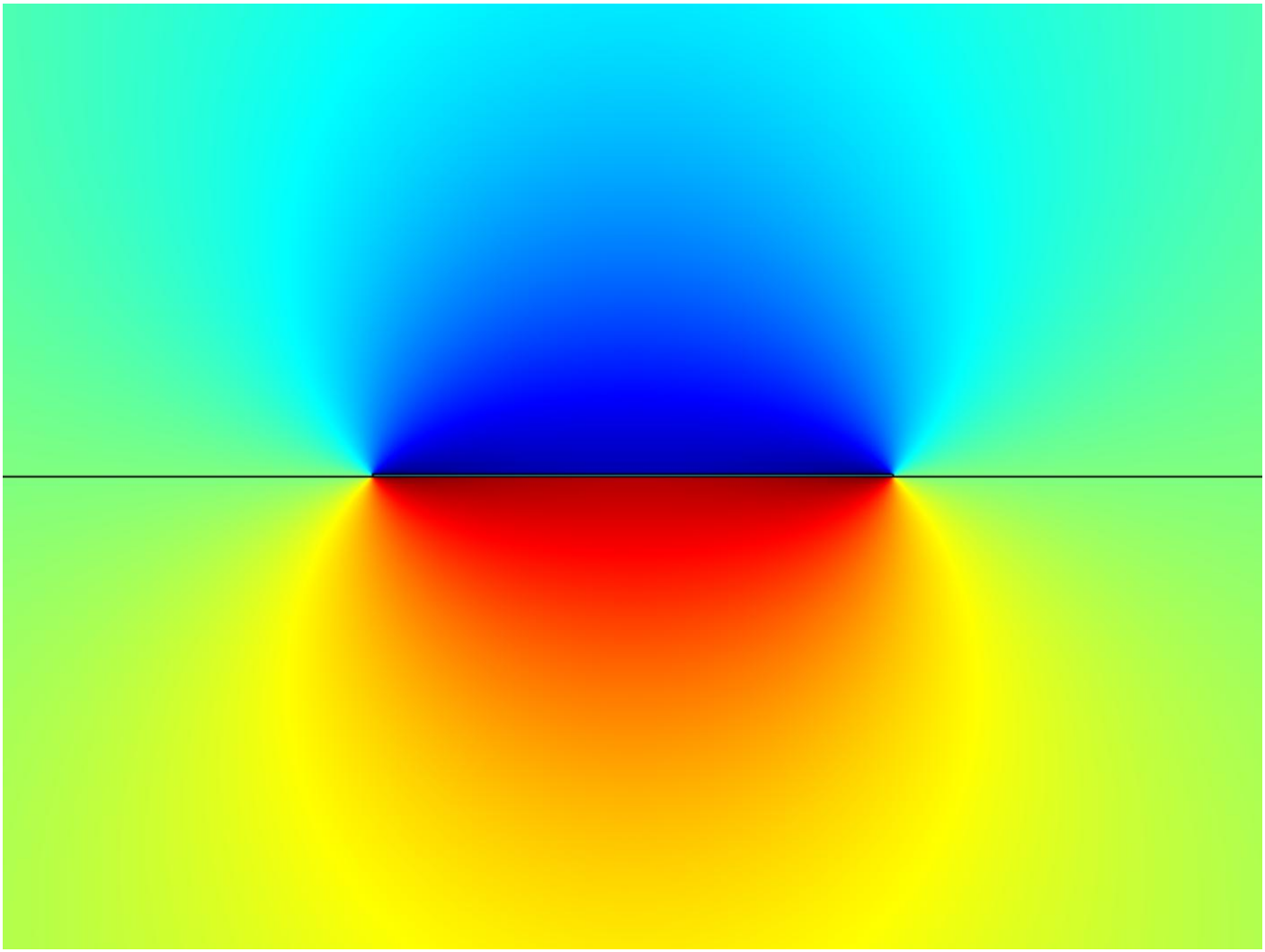


1.6.2 COMSOL Images

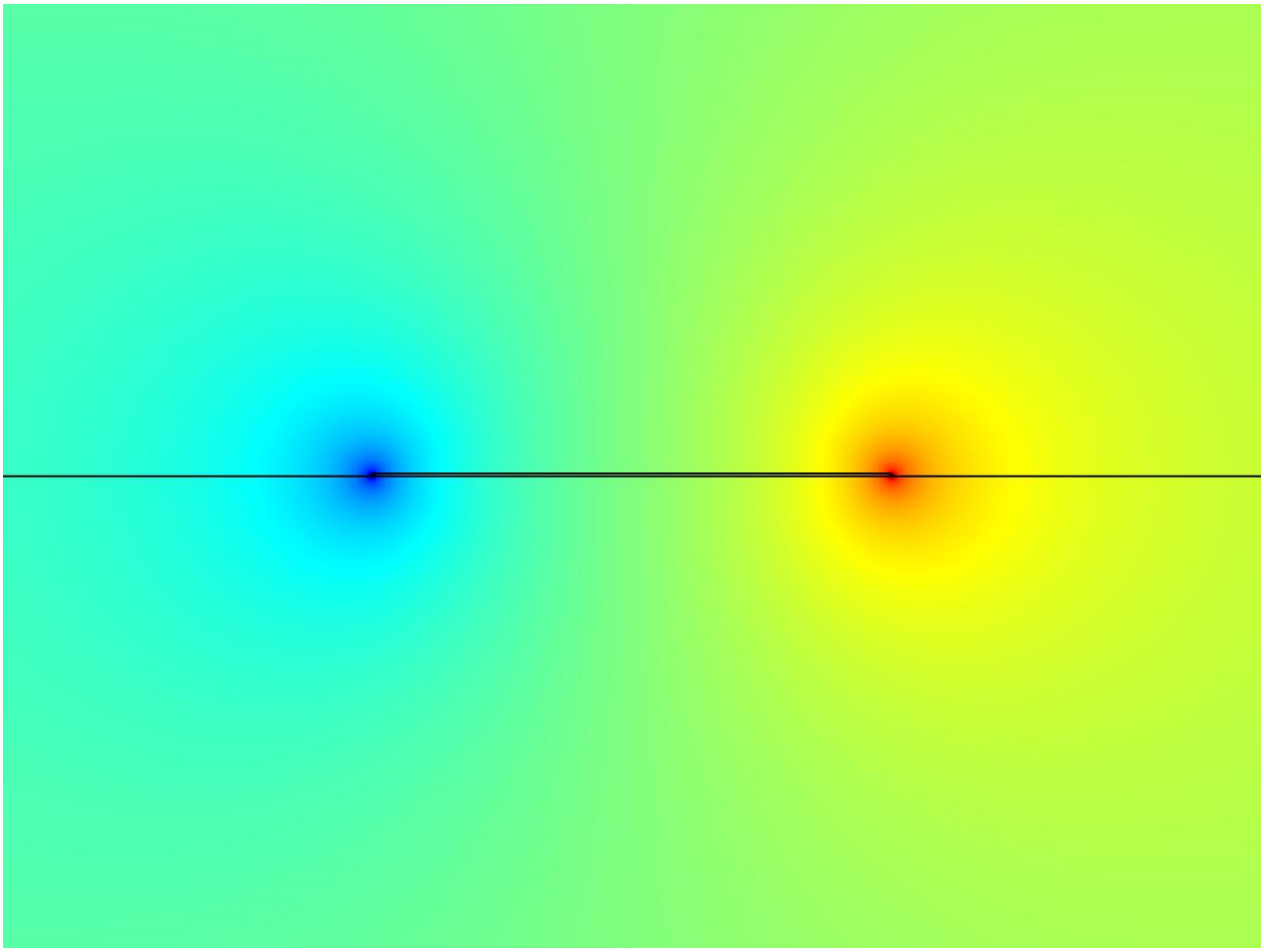
1.) Normal vacuum magnetic flux density



2.) Bx vacuum magnetic flux density

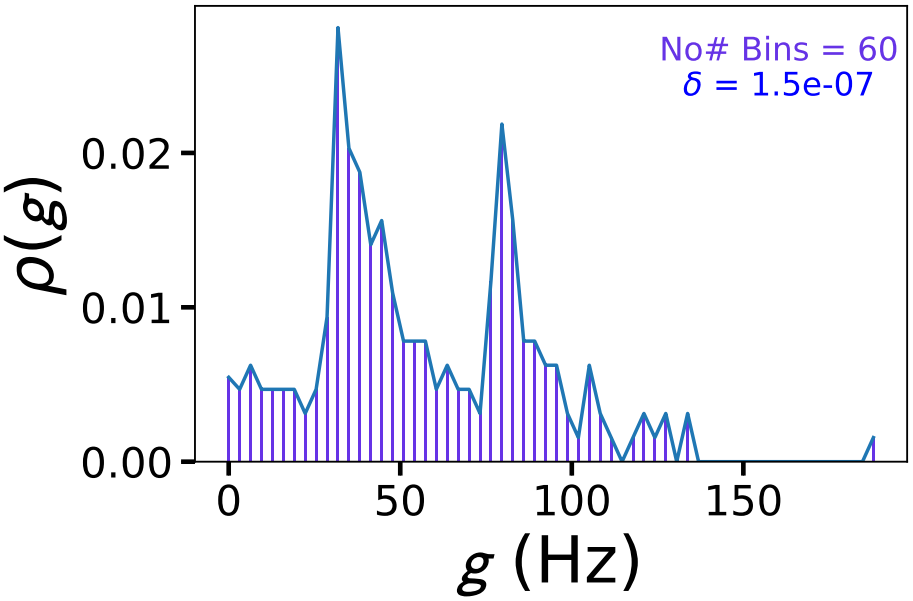


3.) By vacuum magnetic flux density

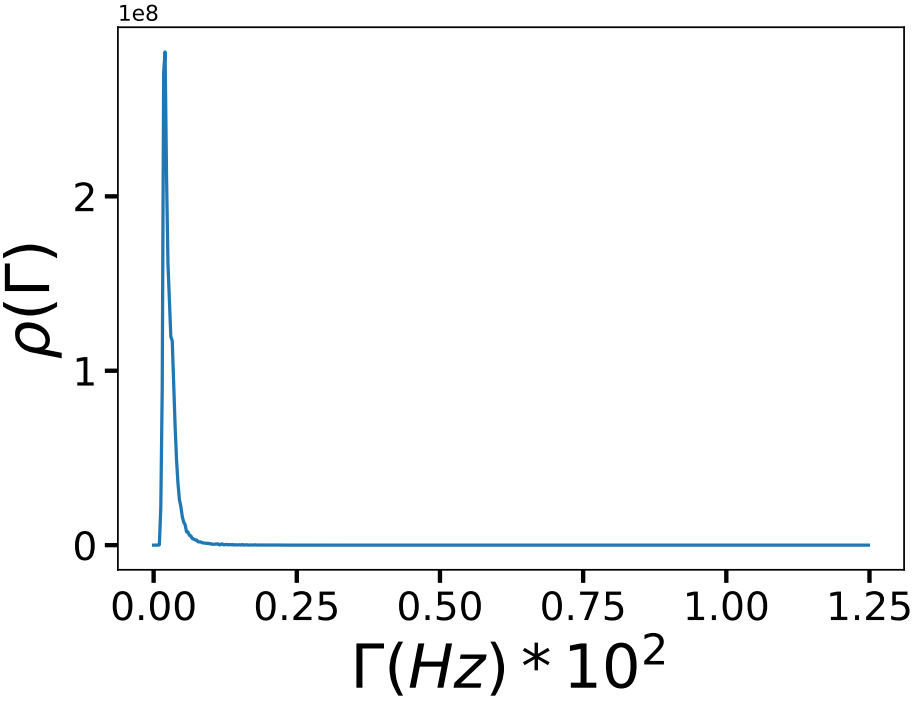


1.6.3 Post-Processing

1.) Spin density



2.) Purcell enhancement



3.) Pulse fidelity

1.7 Resources

Attached are a nnumber of resources relevant to this project, including literature and COMSOL files

1.7.1 COMSOL Files

The CPW is modelled as a two-dimension wire

1.7.2 Literature

1.8 License

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.9 Contact

gareth.jones.16@ucl.ac.uk

London Centre for Nanotechnology, Department of Physics and Astronomy, University College London

1.10 Help

If you are having difficulties with using this package, please email me at gareth.jones.16@ucl.ac.uk

CHAPTER 2

Package Library Classes

2.1 electromagnetics

This package contains the methods for calculating the electromagnetic properties of a cpw resonator. Of particular interest is the spatial distribution of the vacuum current fluctuations for a cpw of a given geometry.

This program wil calculate the electromagnetic properties of cpw resonators

class qsd.electromagnetics.cpw.**CPW**(*x*, *l*, *w*, *t*, *pen*, *Z*, *omega*)

CPW contains the methods for calculating electromagntic properties of cpw resonator

E()

Calculates electroc field of supercodnuctor

J()

Calculates un-normalized vacuum current density

conductivity()

Calculates conductivity of superconductor

current(**args*, ***kwargs*)

Calculates critical current

normalize_J()

Normalizes vacuum current density

resistance()

calculates resistance of superconductor

resistivity()

Calculates resistivity of superconductor

2.1.1 cpw

```
#!/usr/bin/env python

""" This program wil calculate the electromagnetic properties of cpw resonators
"""

import numpy as np
import csv
from scipy import constants as sp

class CPW:
    """
    CPW contains the methods for calculating electromagntic properties of cpw␣
↪resonator
    """
    version = '0.1'

    def __init__(self,x,l,w,t,pen,Z,omega):
        """
        Initializes the cpw geometry, penetration depth, Z, resonant frequency, and␣
↪voltage

        :type x: float
        :param x: width of substrate

        :type l: int
        :param l: length of superconductor

        :type w: int
        :param w: width of superconductor

        :type t: int
        :param t: thickness of superconductor

        :type pen: int
        :param pen: penetration depth of superconductor

        :type Z: int
        :param Z: characteristic impedance or cpw

        :type omega: int
        :param omega: resonant frequency

        :type V: int
        :param l: voltage

        """
        self.x = x
        self.l = l
        self.w = w
        self.t = t
        self.pen = pen
        self.Z = Z
        self.omega = omega
        self.V = 1
```

(continues on next page)

(continued from previous page)

```
    def J(self):
        """
        Calculates un-normalized vacuum current density
        """
        ans = []
        for i in self.x:
            if abs(i) > self.w/2.:
                ans.append(0)
            elif abs(i) == self.w/2.:
                ans.append(1.165/self.pen*(self.w*self.t)**.5)
            elif abs(i) < self.w/2. and abs(i) > self.w/2. - self.pen**2/(2*self.t):
                ans.append(1.165/self.pen*(self.w*self.t)**.5*np.exp(-(self.w/2. -␣
↪abs(i))*self.t/self.pen**2))
            else:
                ans.append((1 - (2*abs(i)/self.w)**2)**-.5)
        return np.asarray(ans)

    def normalize_J(self):
        """
        Normalizes vacuum current density
        """
        #normalise
        Js = self.J()
        dI = self.omega*(sp.hbar/(2*self.Z))**.5
        dx = self.x[1] - self.x[0]
        Jnorm = dI*Js/(self.t*dx*np.sum(Js))
        return Jnorm

    def current(self,*args,**kwargs):
        """
        Calculates critical current
        """
        norm = kwargs.get('norm','yes')

        if norm=='yes':
            I = self.l * self.normalize_J()
        else:
            I = self.l * self.J()
        return I

    def resistance(self):
        """
        calculates resistance of superconductor
        """
        R = self.V / self.current()
        return R

    def resistivity(self):
        """
        Calculates resistivity of superconductor
        """
        A = self. w * self.t # Cross-sectional area of cpw
        rho = self.resistance() * (A / self.l)
        return rho

    def conductivity(self):
        """
```

(continues on next page)

(continued from previous page)

```
        Calculates conductivity of superconductor
        """
        G = 1/self.resistivity()
        return G

    def E(self):
        """
        Calculates electroc field of supercodnuctor
        """
        E = self.normalize_J() * self.conductivity()
        return E
```

2.2 ssh_control

sshcommand includes all of the necessary functions to interface with remote machines via the ssh protocol

class qsd.ssh_control.sshcommand.**SSHCommand**(*host*, **args*, ***kwargs*)

SSHCommand contains the necessary functions to generate bash files which create the ssh interfaces to remote machines

add_remote_machine()

Add a new remote machine to your ssh config file, and gennerate a secure key to share bnetyween your computer and the host machine.

call_bash(*filename*)

Method to call created bash scripts with the subprocess module

check_host_machine()

Check the file structure of the host machine to make sure that the necessary structure is created. If it hasn;t been previously, this script will create the necessary files and folders.

get_comsol_data()

Retrieve exported data from remote machine

job()

Creates a batch job script on the remote machine to run COMSOL

run_comsol()

Run COMSOL on the remote machine

upload_data()

Upload data to the remote machine

upload_job_script()

Upload the batch job script to the remote machine

2.2.1 sshcommand

```
#!/usr/bin/env python
"""
sshcommand includes all of the necessary functions to interface with remote machines␣
↪via the ssh protocol
"""
```

(continues on next page)

(continued from previous page)

```
from subprocess import call
import os
import stat

class SSHCommand:
    """
        SSHCommand contains the necessary functions to generate bash files which␣
↪create the ssh interfaces to remote machines
    """
    def __init__(self,host,*args,**kwargs):
        """
            :type: host = str
            :param: host = name of the machine you want to connect to

            :type: host_network str
            :param: host_network = network address of the remote machine

            :type: full_host = str
            :param: full_host = full network address, including host

            :type: user = str
            :param: user = username

            :type: model = str
            :param: model = COMSOL model name

            :type: paramfile = str
            :param: paramfile = file containing parameters for COMSOL simulation

        """
        self.host = host
        self.host_network = kwargs('host_network',None)
        self.full_host = self.host + "@" + self.full_host
        self.user = kawrgs.get('user',None)
        self.model = kwargs.get('model','cpw_vacuum_calcs.mph')
        self.paramfile = kwargs.get('paramfile',None)
        return

    def add_remote_machine(self):
        """
            Add a new remote machine to your ssh config file, and gennerate a secure␣
↪key to share bnetyween your computer and the host machine.
        """
        file = open("keygen","w")
        file.write("#!/bin/bash")
        file.write("cd ~/.ssh\n")
        file.write("ssh-keygen\n")
        file.write("scp id_rsa %s:/~/.ssh/authorized-keys\n" % self.full_host)
        file.write("cd -\n")
        file.close()
        self.call_bash('keygen')

        sshfile = os.getenv("HOME") + '/.ssh/config'
        file = open(sshfile,"a")
        file.write("\n")
        file.write("Host %s\n" % self.host)
        file.write("    HostName %s\n" % self.host_network)
```

(continues on next page)

(continued from previous page)

```
        file.write("    User %s\n" % self.user)
        file.write("    IdentityFile ~/.ssh/id_rsa")
        file.close()
        self.call_bash(sshfile)

    def upload_data(self):
        """
            Upload data to the remote machine
        """
        print("Uploading data to remote machine")

        filename = "upload_data"
        filedir = os.getcwd() + "/" + filename
        file = open(filename,"w")
        file.write("#!/bin/bash\n")
        file.write("\n")
        file.write('filename = "set_comsol_data"\n')
        file.close()
        self.call_bash(filename)

    def run_comsol(self):
        """
            Run COMSOL on the remote machine
        """
        print("Running COMSOL on remote machine...")

        filename = "run_comsol"
        filedir = os.getcwd() + "/" + filename
        file = open(filedir,"w")
        file.write("#!/bin/bash\n")
        file.write("\n")
        file.write('HOST="%s"\n' % self.host)
        file.write('echo "Running COMSOL on ${HOST}"\n')
        file.write("ssh ${HOST} 'cd COMSOL_files && ./job input/cpw_vacuum_calcs.mph';
↪ exit\n")
        file.close()
        self.call_bash(filename)

    def check_host_machine(self):
        """
            Check the file structure of the host machine to make sure that the␣
↪necessary structure is created. If it hasn;t been previously, this script will␣
↪create the necessary files and folders.
        """
        print("Checking host machine file structure")

        filename = "check_host"
        filedir = os.getcwd() + "/" + filename
        file = open(filedir,"w")
        file.write("#!/bin/bash\n")
        file.write("\n")
        file.write('HOST="%s"\n' % self.host)
        file.write("ssh ${HOST} '[ ! -d \"COMSOL_files\" ] && echo \"Creating remote␣
↪folder structire\"&& mkdir COMSOL_files COMSOL_files/input COMSOL_files/output␣
↪COMSOL_files/exports COMSOL_files/parameter_files;  exit'\n")
        file.write("scp %s ${HOST}:~/COMSOL_files/input/\n'"  % self.model)
        file.close()
```

(continues on next page)

(continued from previous page)

```
        self.call_bash(filename)

    def set_comsol_data(self):
        print("Uploading comsol parameter files...")
        filename = "set_comsol_data"
        filedir = os.getcwd() + "/" + filename

        file = open(filedir,"w")
        file.write("#!/bin/bash\n")
        file.write("\n")
        file.write('MODELNAME="%s"\n' % self.model)
        file.write('PARAMFILE="${MODELNAME}.txt"\n')
        file.write('cp "%s" ${PARAMFILE}\n' % self.paramfile)
        file.write('scp ${PARAMFILE} %s:~/COMSOL_files/parameter_files\n' % self.host)
        #file.write('scp ${PARAMFILE} %s:/homes/gjones/COMSOL_files/parameter_files\n
↪' % self.host)
        file.write("\n")
        file.write('rm ${PARAMFILE}\n')
        file.close()
        self.call_bash(filename)

    def get_comsol_data(self):
        """
            Retrieve exported data from remote machine
        """
        print("Retrieving comsol datafiles...")

        down_dir = os.path.join(os.getcwd(), "downloads")
        if not os.path.exists(down_dir):
            os.mkdir(down_dir)

        filename = "get_comsol_data"
        filedir = os.getcwd() + "/" + filename
        file = open(filename,"w")
        file.write("#!/bin/bash\n")
        file.write("\n")
        file.write('HOST="gade"\n')
        file.write('REMOTEDIR="COMSOL_files/exports"\n')
        file.write('DOWNLOADDIR="%s"\n' % down_dir)
        file.write('scp -r ${HOST}:${REMOTEDIR} ${DOWNLOADDIR}\n')
        file.close()
        self.call_bash(filename)

    def upload_job_script(self):
        """
            Upload the batch job script to the remote machine
        """
        self.job()
        print("Uploading job script....")
        filename = "upload_job"
        filedir = os.getcwd() + "/" + filename
        file = open(filename,"w")
        file.write("#!/bin/bash\n")
        file.write("\n")
        file.write("chmod +xu job\n")
        file.write('scp job %s:~/COMSOL_files\n' %self.host)
        file.close()
```

(continues on next page)

(continued from previous page)

```
        self.call_bash(filename)
        os.remove('job')

    def job(self):
        """
            Creates a batch job script on the remote machine to run COMSOL
        """
        filename = "job"
        filedir = os.getcwd() + "/" + filename
        file = open(filename,"w")
        file.write("#!/bin/bash\n")
        file.write("\n")
        file.write("MODELTOCOMPUTE=$@\n")
        file.write("INPUTFILE=\"${HOME}/COMSOL_files/${MODELTOCOMPUTE}\"\n")
        file.write("PARAMFILE=\"${HOME}/COMSOL_files/parameter_files/${MODELTOCOMPUTE#
↪'input/'}.txt\"\n")
        file.write("OUTPUTFILE=\"${HOME}/COMSOL_files/output/${MODELTOCOMPUTE#'input/
↪'}\"\n")
        file.write("BATCHLOG=\"${HOME}/COMSOL_files/logs/${MODELTOCOMPUTE}.log\"\n")
        file.write("JOB=\"b1\"\n")
        file.write("\n")
        file.write("# Get the parameters from the text file\n")
        file.write("declare -a NAMEARRAY VALUEARRAY DESCARRAY\n")
        file.write("let i=0 \n")
        file.write("while IFS=\" \" read -r name value description \n")
        file.write("do\n")
        file.write("   NAMEARRAY[i]=\"${name}\"\n")
        file.write("   VALUEARRAY[i]=\"${value}\"\n")
        file.write("   DESCARRAY[i]=\"${description}\"\n")
        file.write("   ((++i))\n")
        file.write("done < ${PARAMFILE}\n")
        file.write("\n")
        file.write("# Concatenate string to remove whitespace and add commas after␣
↪each element\n")
        file.write("NAMES=$(IFS=, eval 'echo \"${NAMEARRAY[*]}\"')\n")
        file.write("VALUES=$(IFS=, eval 'echo \"${VALUEARRAY[*]}\"')\n")
        file.write("DESC=$(IFS=, eval 'echo \"&{DESCARRAY[*]}\"')\n")
        file.write("\n")
        file.write("# run comsol directly from the command line. requires a user␣
↪input for the input file\n")
        file.write("comsol batch -inputfile ${INPUTFILE} -outputfile ${OPUTPUTFILE} -
↪pname ${NAMES} -plist ${VALUES} -job ${JOB}\n")
        file.write("mv on.* output/\n")

    def call_bash(self,filename):
        """
            Method to call created bash scripts with the subprocess module
        """
        st = os.stat(filename)
        os.chmod(filename, st.st_mode | stat.S_IEXEC)
        callname = './'+filename
        rc = call(callname)
        os.remove(filename)
```

2.3 data_processing

2.3.1 preproc

Method to upload data to remote machine

class qsd.data_processing.preproc.**PreProc**
Preprocessing methods

upload_data()
Upload data to remote machine

```
#!/usr/bin/env python
"""
Method to upload data to remote machine
"""
from subprocess import call

class PreProc:
    """
    Preprocessing methods
    """
    def upload_data(self):
        """
        Upload data to remote machine
        """
        rc = call("./upload_data")
```

2.3.2 setparams

This program sets the parameters required for simulation.

class qsd.data_processing.setparams.**SetParams**
This class allows a user to set the relevant parameters of the cpw. This needs refactorinng, but is sufficient for now.

param_list(*x*, *I*, *Jnorm*, *paramfile*)
Generates a text file which holds the parameters requiured for the COMSOL simulation

set_params(*infile*)
Returns simulation parameters as a dictionary

```
#!/usr/bin/env python
"""
This program sets the parameters required for simulation.
"""

import numpy as np
import os
from subprocess import call

class SetParams:
    """
        This class allows a user to set the relevant parameters of the cpw. This␣
↪needs refactorinng, but is sufficient for now.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        """
            Initialize with parameters

            :type w: float
            :param w: width of substrate

            :type t: float
            :param t: thickness of superconductor

            :type l: float
            :param l: length of supercondducting wire

            :type pen: float
            :param pen: penetration depth

            :type omega: float
            :param omega: cavity resonant frequency

            :type Z: float
            :param w: characteristic impedance
        """
        self.w = None
        self.t = None
        self.l = None
        self.pen = None
        self.omega = None
        self.Z = None
        self.N = 2

    def set_params(self,infile):
        """
            Returns simulation parameters as a dictionary
        """
        # Define geometry of the superconductor
        paramfile=open(infile,"r")
        filestring = paramfile.read()
        filelist = filestring.split("\n")

        pd = {}
        for fl in filelist:
            l = fl.split()
            pd[l[0]] = l[2]
        paramfile.close()

        self.w = float(pd["w"])
        self.t = float(pd["t"])
        self.l = float(pd["l"])
        self.pen = float(pd["pen"])
        self.omega = float(pd["omega"])
        self.Z = float(pd["Z"])

        params = {'w':self.w,
            't':self.t,
            'l':self.l,
            'pen':self.pen,
            'omega':self.omega,
```

(continues on next page)

(continued from previous page)

```
            'Z':self.Z
        }
        return params

    def param_list(self,x,I,Jnorm,paramfile):
        """
        Generates a text file which holds the parameters requiured for the COMSOL␣
↪simulation
        """
        n = [abs(i) for i in x]
        idx = n.index(min(n))

        I0 = I[idx]
        J0 = I0/(2*(self.w+self.t)*self.pen)
        pen_perp = self.pen**2 / (2*self.t)
        C = (0.506*np.sqrt(self.w/(2*pen_perp)))**0.75
        l1 = self.pen*np.sqrt(2*self.pen/pen_perp)
        l2 = 0.774*self.pen**2/pen_perp + 0.5152*pen_perp
        J2overJ1 = (1.008/np.cosh(self.t/self.pen)*np.sqrt(self.w/pen_perp/
            (4*pen_perp/self.pen - 0.08301*self.pen/pen_perp)))
        J1 = Jnorm[idx]
        w_sub = 4*self.w
        h_sub = 25e-06

        f = open(paramfile,'w')
        f.write('w ' + str(self.w) + '[m] width_of_superconductor\n'
           't ' + str(self.t) + '[m] thickness_of_superconductor\n'
           'pen ' + str(self.pen) + '[m] penetration_depth\n'
           'I0 ' + str(I0) + '[A/m] current_at_x=0\n'
           'J0 ' + str(J0) + '[A/m^3] current_density_at_x=0\n'
           'N ' + str(self.N) + '\n'
           'w_sub ' + str(w_sub) + '[m] substrate_width\n'
           'h_sub ' + str(h_sub) + '[m] substrate_height\n'
           'pen_perp ' + str(pen_perp) + '[m] perpendicular_pen_depth\n'
           'C ' + str(C) + ' capacitance\n'
           'l1 ' + str(l1) + '[m]\n'
           'l2 ' + str(l2) + '[m]\n'
           'J2overJ1 ' + str(J2overJ1) + '\n'
           'J1 ' + str(J1) + '[A/m]'
           )

        f.close()
```

2.3.3 readcomsol

Methods for reading data from COMSOL

class qsd.data_processing.readcomsol.**ReadComsol**(*file*)

ReadComsol contains methods for reading datafiles from COMSOL

read_1D_comsol_data()

Read 1D COMSOL datafiles

read_2D_comsol_data()

Read 2D COMSOL datafiles

read_full_data()
Read full COMSOL datafiles

```
#!/usr/bin/env python
"""
    Methods for reading data from COMSOL
"""

import numpy as np
import csv
from subprocess import call

class ReadComsol:
    """
        ReadComsol contains methods for reading datafiles from COMSOL
    """

    def __init__(self,file):
        """
            Initialize with COMSOL file
        """
        self.file = file

    def read_1D_comsol_data(self):
        """
            Read 1D COMSOL datafiles
        """
        x=[]
        y=[]
        with open(self.file, 'r') as rf:
            reader = csv.reader(rf, delimiter=',')
            for row in reader:
                x.append(row[0])
                y.append(row[1])
        x = np.asarray((x),dtype=float)
        y = np.asarray((y),dtype=float)
        return x,y

    def read_2D_comsol_data(self):
        """
            Read 2D COMSOL datafiles
        """
        x=[]
        y=[]
        z=[]
        with open(self.file, 'r') as rf:
            reader = csv.reader(rf, delimiter=',')
            for row in reader:
                x.append(row[0])
                y.append(row[1])
                z.append(row[2])
        x = np.asarray((x),dtype=float)
        y = np.asarray((y),dtype=float)
        z = np.asarray((z),dtype=float)
        return x,y,z

        def read_full_data(self):
        """
```

(continues on next page)

(continued from previous page)

```
            Read full COMSOL datafiles
        """
        x=[]
        y=[]
        z=[]
        with open(self.file, 'r') as rf:
            reader = csv.reader(rf, delimiter=',')
            for row in reader:
                x.append(row[0])
                # Remove header from csv file, if it exists
                if x[0].split()[0] == '%':
                    x.remove(row[0])
                else:
                    y.append(row[1])
                    z.append(row[2])
        return x,y,z
```

2.3.4 postproc

This program allows a user to determine certain figures of merit of interest for cpw resonators for quantum spin dynamics.

class qsd.data_processing.postproc.**PostProc**(*w*, *t*, *l*, *pen*, *omega*, *Z*)

Contains methods for calculating various figures of merit for cpw

B1(*dbx*, *dby*, *theta*)

Calculates total B1 field

average_photon_number()

Calculates average photon number

cooperativity()

Calculates cooperativity

coupling(*dbx*, *dby*, **args*, ***kwargs*)

Calculates coupling constant g, <m|Sx|m> * ue * sqrt(dby^2 + cos(theta) dbx^2)

cut_line_single_spin_coupling(*Bx*, *By*, **args*, ***kwargs*)

Calculates the single spin coupling for a given grid area

cut_line_spin_density(*g*)

Calculates the spin density for cut line section

distribution(*x*, *y*, *param*, **args*, ***kwargs*)

Method to calculate histogram

finesse()

Calculates finesse

larmor_density(*x*, *y*, *theta_larmor*, **args*, ***kwargs*)

Calculates distribution of Larmor frequency

larmor_omega(*B*, *gamma*)

Calculates larmor precession frequency

larmor_theta(*omega_larmor*, *tau*)

Calculates angle of larmor precession

ncell (*x*, *y*, *param*)
Number of cells in resonator

purcell_density (*x*, *y*, *gamma*, **args*, ***kwargs*)
Calculates distribution of purcell rate in resomator

purcell_factor (*lambda_c*, *n*, *Q*)
Calculates the Purcell enhancement induced by the cavity

purcell_rate (*g*, *Q*, **args*, ***kwargs*)
Calculates the Purcell rate

spin_density (*x*, *y*, *g*, **args*, ***kwargs*)
Calculates distribution of spins in resonator

spinmap (*xin*, *yin*, *spin_depth*)
Defines the layer at which spins are implanted. At the moment, only specified for bulk doping

```
#!/usr/bin/env python

"""
This program allows a user to determine certain figures of merit of interest for cpw␣
↪resonators for quantum spin dynamics.
"""

import numpy as np
import numpy.matlib
from scipy import constants as sp
from qsd.data_processing import setparams

class PostProc:
    """
    Contains methods for calculating various figures of merit for cpw
    """
    def __init__(self,w,t,l,pen,omega,Z):
        """
        Initializes resonator structure
        """
        #setp = setparams.SetParams()
        #params = setp.set_params()
        #self.w = params["w"]
        #self.t = params["t"]
        #self.l = params["l"]
        #self.pen = params["pen"]

        #define the resonator - from CST or experiment
        #self.omega = params["omega"]
        #self.Z = params["Z"]
        self.w = w
        self.t = t
        self.l = l
        self.pen = pen
        self.omega = omega
        self.Z = Z

        self.g = None

        self.volume_cell = None
```

(continues on next page)

(continued from previous page)

```
    def B1(self,dbx,dby,theta):
        """
        Calculates total B1 field
        """
        #B1 = np.sqrt(dby**2 + (np.cos(theta)**2)*dbx**2)
        B1 = dbx + dby
        return B1

    def larmor_omega(self,B,gamma):
        """
        Calculates larmor precession frequency
        """
        omega_larmor = gamma * B
        return omega_larmor

    def larmor_theta(self,omega_larmor,tau):
        """
        Calculates angle of larmor precession
        """
        theta_larmor = omega_larmor * tau
        return theta_larmor

    def cut_line_single_spin_coupling(self,Bx,By,*args,**kwargs):
        """
        Calculates the single spin coupling for a given grid area
        """
        theta = kwargs.get('theta',0)
        ang = np.cos(theta)
        ue = sp.physical_constants["Bohr magneton"][0]
        self.g = 0.47 * ue * np.sqrt(By**2 + (ang**2) * Bx**2)
        return self.g/sp.h

    def cut_line_spin_density(self,g):
        """
        Calculates the spin density for cut line section
        """
        self.volume_cell = g * self.t * self.l
        rho =  sp.m_e / self.volume_cell
        return rho

    def distribution(self,x,y,param,*args,**kwargs):
        """
        Method to calculate histogram
        """
        bin_num = kwargs.get('bins',500)
        Ncell = self.ncell(x,y,param)
        param = np.matlib.repmat(param, 1, Ncell)

        # Calculate histogram
        hist, edges = np.histogram(param, bins=bin_num) # single spin
        hist = hist * Ncell # with 3d box
        hist = hist / sum(hist) # normalize
        edges = edges[0:len(hist)] # shift bin edges to get the same length as data
        return hist, edges

    def spin_density(self,x,y,g):
        """
```

(continues on next page)

(continued from previous page)

```
        Calculates distribution of spins in resonator
        """
        hist, edges = self.distribution(x,y,g)

        return hist, edges

    def larmor_density(self,x,y,theta_larmor):
        """
        Calculates distribution of Larmor frequency
        """
        hist, edges = self.distribution(x,y,theta_larmor)
        return hist, edges

    def purcell_density(self,x,y,gamma):
        """
        Calculates distribution of purcell rate in resomator
        """
        hist, edges = self.distribution(x,y,gamma)
        return hist, edges

    def ncell(self,x,y,param):
        """
        Number of cells in resonator
        """
        # Reshape g so we can append multiple values for each box section
        param=param.reshape(len(param),1)

        # Calculate the size of the boxes
        bucket = x.count(x[0]) # number of samples for each point in space
        x_box = abs(float(x[bucket-1]) - float(x[bucket]))
        y_box = abs(float(y[0]) - float(y[1]))
        z_box = x_box
        volume = x_box * y_box * z_box

        # Calculate number of spins in each cell
        no_spins_in_box = 1e7
        Ncell = round(no_spins_in_box * volume)
        return Ncell

    def purcell_rate(self,g,Q,*args,**kwargs):
        """
        Calculates the Purcell rate
        """
        k = self.omega / Q
        omega_s = kwargs.get('omega_s',self.omega)
        delta = self.omega - omega_s

        purcell = k * ((g**2) / (k**2) / (4 + delta**2))
        # purcell = (4*(g**2)) / k
        return purcell

    def purcell_factor(self,lambda_c,n,Q):
        """
        Calculates the Purcell enhancement induced by the cavity
        """
        F = ( 3 / (4*np.pi**2) ) * (lambda_c / n)**3 * ( Q / (self.w * self.t * self.
↪l))
```

(continues on next page)

(continued from previous page)

```
        return F

    def coupling(self,dbx,dby,*args,**kwargs):
        """
        Calculates coupling constant g, <m|Sx|m> * ue * sqrt(dby^2 + cos(theta) dbx^2)
        """
        theta = kwargs.get('theta',0) # Angle the static magnetic field is applied on
        ang = np.cos(theta)
        ue = sp.physical_constants["Bohr magneton"][0]
        g = [*map(lambda x,y: 0.47 * ue * np.sqrt(y**2 + x**2),dbx,dby)]
        g = np.asarray([x / sp.h for x in g])
        return g

    def average_photon_number(self):
        """
        Calculates average photon number
        """
        n = (4 * k1 * Pin) / (sp.hbar * self.omega * (k1 + k2 + kL)**2)
        return n

    def cooperativity(self):
        """
        Calculates cooperativity
        """
        return

    def finesse(self):
        """
        Calculates finesse
        """
        return
```

CHAPTER 3

Examples

The following examples show how to use the package in order to generate a cpw geometry, run the calculations on remote machines, and process the data for various figures of merit.

3.1 Preprocessing - Determine the Current Density

The first thing which is required is to define the geometry of the cpw. In this code exapmle, we call the SetParams object, which reads a text file containing the relevant resonator parameters. We define a grid over the enntire resonator structure, and then calcualte analytically the current density and critical current for the cpw. These values are saved in a parameter listr which gets sent to COMSOL, and the datafiles are stored locally.

```
import csv
import os
import numpy as np
from qsd.electromagnetics import cpw
from qsd.data_processing import setparams

# Define geometry of the superconductor
setp = setparams.SetParams()
params = setp.set_params("cpw_parameters.txt")

w = params["w"]
t = params["t"]
l = params["l"]
pen = params["pen"]
omega = params["omega"]
Z = params["Z"]

# Define the 'mesh'
x = np.linspace(-w, w, int(1e04))

# Instantiate Special CPW object
cpw = cpw.CPW(x,l,w,t,pen,Z,omega)
```

(continues on next page)

(continued from previous page)

```
Js = cpw.J() #s Current density - not normalised
Jnorm = cpw.normalize_J() # Normalise
I = cpw.current(norm='no') # Find the current

# Generate a parameter list for COMSOL modelling
paramlist = setp.param_list(x,I,Jnorm,'paramlist.txt') # Generate COMSOL parameter␣
↪list

currentDensityFile = str(os.getcwd() + "/current_density.csv")
np.savetxt(currentDensityFile, np.column_stack((x,Jnorm)), delimiter=",")

currentFile = str(os.getcwd() + "/current.csv")
np.savetxt(currentFile, np.column_stack((x,I)), delimiter=",")
```

3.2 Adding a remote machine

If you want to use ssh protocols to interface with different machines, it is often easier to set up your computers ssh config file such that you have an authenticated connection to a host machine that you recognise. This is achieved by genberating a secure public key which is shared between your machine and the host, and creating an alias so that you can loginn to the remote machine without having to type out the full host name each time. The method 'add_remote_machine' in ssh_control does this for you. Enter the name of the host machine you want to connect to (e.g. monaco/viena/etc) along with the network address and your username, and the rest will be taken care of.

```
#!/usr/bin/env python

from qsd import ssh_control

host_machine = 'monaco'
network = '.ee.ucl.ac.uk'
username = 'ucapxxx'

# Instantiate the ssh object
sshc = ssh_control.sshcommand()

# Add the remote machine to you ssh config file in ~.ssh
sshc.add_remote_machine(host_machine, host_network=network, user=username)
```

3.3 Interfacing with a Remote Machine

```
#!/usr/bin/env python
from qsd.ssh_control import sshcommand

# Specify remote computer name
host = 'monaco'

sshc = sshcommand.SSHCommand(host,model='cpw_vacuum_calcs.mph',paramfile='paramlist.
↪txt')

# Securely copy the parameter list to remote machine
#sshc.scp_params()
```

(continues on next page)

(continued from previous page)

```
# Ensure that the remote machine has the folder structure
sshc.check_host_machine()

# Copy the parameter file to correct directory
sshc.set_comsol_data()

sshc.upload_job_script()

# Run COMSOL on remote machine
sshc.run_comsol()

# Download data from remote machine
sshc.get_comsol_data()
```

3.4 Postprocessing - Single Spin Coupling

```
import os
import numpy as np
from qsd.data_processing import readcomsol,postproc,setparams

# Read data from downloads
file_dbx = os.getcwd() + '/downloads/exports/Bx_fullData.csv'
file_dby = os.getcwd() + '/downloads/exports/By_fullData.csv'

rdx = readcomsol.ReadComsol(file_dbx)
rdy = readcomsol.ReadComsol(file_dby)

# Read csv file, and get x,y annd dbx/dby data for each
# blocked point in space
bx_x,bx_y,bx_z = rdx.read_full_data()
by_x,by_y,by_z = rdy.read_full_data()

dbx = np.asarray(bx_z).astype(np.float)
dby = np.asarray(by_z).astype(np.float)

# Define geometry of the superconductor
setp = setparams.SetParams()
params = setp.set_params("cpw_parameters.txt")

w = params["w"]
t = params["t"]
l = params["l"]
pen = params["pen"]
omega = params["omega"]
Z = params["Z"]

# Postprocess data
post = postproc.PostProc(w,t,l,pen,omega,Z)

# Single spin coupling for each point on mesh grid
g = post.coupling(dbx,dby,theta=0)
hist, edges = post.spin_density(bx_x,bx_y,g) # density
```

3.5 Postprocessing - Purcell Enhancement

```
#!/usr/bin/env python

from scipy import constants as sp
import os
import numpy as np
from matplotlib import pyplot as plt
from qsd.data_processing import readcomsol,postproc,setparams

# Read data from downloads
file_dbx = os.getcwd() + '/downloads/exports/Bx_fullData.csv'
file_dby = os.getcwd() + '/downloads/exports/By_fullData.csv'

rdx = readcomsol.ReadComsol(file_dbx)
rdy = readcomsol.ReadComsol(file_dby)

# Read csv file, and get x,y annd dbx/dby data for each
# blocked point in space
bx_x,bx_y,bx_z = rdx.read_full_data()
by_x,by_y,by_z = rdy.read_full_data()

dbx = np.asarray(bx_z).astype(np.float)
dby = np.asarray(by_z).astype(np.float)

# # Define geometry of the superconductor
setp = setparams.SetParams()
params = setp.set_params("cpw_parameters.txt")

w = params["w"]
t = params["t"]
l = params["l"]
pen = params["pen"]
omega = params["omega"]
Z = params["Z"]

# Postprocess data
post = postproc.PostProc(w,t,l,pen,omega,Z)

# Single spin coupling for each point on mesh grid
g = post.coupling(dbx,dby,theta=0)
hist, edges = post.spin_density(bx_x,bx_y,g) # density

plt.plot(edges,hist)
plt.show()
```

3.6 Postprocessing - Pi Pulse Fidelity

```
#!/usr/bin/env python

from scipy import constants as sp
import os
import numpy as np
from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

```
from qsd.data_processing import readcomsol,postproc,setparams

# Read data from downloads
file_dbx = os.getcwd() + '/downloads/exports/Bx_fullData.csv'
file_dby = os.getcwd() + '/downloads/exports/By_fullData.csv'

rdx = readcomsol.ReadComsol(file_dbx)
rdy = readcomsol.ReadComsol(file_dby)

# Read csv file, and get x,y annd dbx/dby data for each
# blocked point in space
bx_x,bx_y,bx_z = rdx.read_full_data()
by_x,by_y,by_z = rdy.read_full_data()

dbx = np.asarray(bx_z).astype(np.float)
dby = np.asarray(by_z).astype(np.float)

# # Define geometry of the superconductor
setp = setparams.SetParams()
params = setp.set_params("cpw_parameters.txt")

w = params["w"]
t = params["t"]
l = params["l"]
pen = params["pen"]
omega = params["omega"]
Z = params["Z"]

# Postprocess data
post = postproc.PostProc(w,t,l,pen,omega,Z)

# Single spin coupling for each point on mesh grid
g = post.coupling(dbx,dby,theta=0)

# Calculate total B1 field
theta = 0
B1 = post.B1(dbx, dby, theta)

# Calculate Larmor frequency
gamma = 4.32e07 # Bismuth gyromagnetic ratio (rad/T*s)
omega_larmor = post.larmor_omega(B1,gamma)
tau = 1
theta_larmor = post.larmor_theta(omega_larmor, tau)

lardens, laredge = post.larmor_density(bx_x,by_y,theta_larmor)

# Weight theta with contribution to spin signal
g_weight = np.zeros(len(laredge))
for i in range (0,len(laredge)-1):
    g_weight[i] = sum(g[np.where(np.logical_and(theta_larmor>=laredge[i], theta_larmor
↪<=laredge[i+1]))])

rho_weighted = lardens * g_weight**2

plt.plot(laredge,rho_weighted)
plt.show()
```

3.7 Postprocessing - Single Spin Coupling for a Cut Line

```
from qsd.data_processing import readcomsol,postproc
import numpy as np
import os

#read in 1d data from comsol for plotting
bx = readcomsol.ReadComsol(os.getcwd() + '/downloads/exports/Bx.csv')
by = readcomsol.ReadComsol(os.getcwd() + '/downloads/exports/By.csv')
bn = readcomsol.ReadComsol(os.getcwd() + '/downloads/exports/normB.csv')

xx,Bx = bx.read_1D_comsol_data()
xy,By = by.read_1D_comsol_data()
xn,Bn = bn.read_1D_comsol_data()

# Define geometry of the superconductor
setp = setparams.SetParams()
params = setp.set_params("cpw_parameters.txt")

w = params["w"]
t = params["t"]
l = params["l"]
pen = params["pen"]
omega = params["omega"]
Z = params["Z"]

#calcualte single spin couplinng coefficient
pp = postproc.PostProc(w,t,l,pen,omega,Z)
g = pp.cut_line_single_spin_coupling(Bx,By)

rho = pp.cut_line_spin_density(g)
rho = rho / sum(rho)
```

3.8 Postprocessing - Purcell Enhancement for a Cut Line

```
#!/usr/bin/env python
from qsd.data_processing import readcomsol,postproc
import numpy as np
from scipy import constants as sp
import os

#read in 1d data from comsol for plotting
bx = readcomsol.ReadComsol(os.getcwd() + 'downloads/exports/Bx.csv')
by = readcomsol.ReadComsol(os.getcwd() + 'downloads/exports/By.csv')
bn = readcomsol.ReadComsol(os.getcwd() + 'downloads/exports/normB.csv')

xx,Bx = bx.read_1D_comsol_data()
xy,By = by.read_1D_comsol_data()
xn,Bn = bn.read_1D_comsol_data()

# Define geometry of the superconductor
setp = setparams.SetParams()
params = setp.set_params("cpw_parameters.txt")
```

(continues on next page)

(continued from previous page)

```
w = params["w"]
t = params["t"]
l = params["l"]
pen = params["pen"]
omega = params["omega"]
Z = params["Z"]

lambda_c = 6e-03 # Will work out properly, but just testing for now
epsilon_r = 11.9
n = np.sqrt(epsilon_r) / sp.c # Dielectric constant
Q = 20000 # Will get this data from CST
F = pp.purcell_factor(lambda_c,n,Q)
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

q

Index