**Restructuring of HIT-SI Current Equilibrium Code and Construction of Coil GUI**

Zach Malek

September 19th, 2014

# Table of Contents

Table of Contents 2

Introduction 3

Construction of Coil Object and Equations for Coil Fields…………………...…………………………………………………..3

Construction of Coil Set Object…………………...…………………………………………………………………………………………..4

Construction of Coil Simulation Object…………………...………………………………………………………………………………5

Construction of Plasma Coil Object………………………………………………………………………………………………………….6

Writing of the Coil/Floop Placement Script………………………………………………………………………………………………7

Construction of Coil GUI………………………….………………………………………………………………………………………………7

Further Work 7

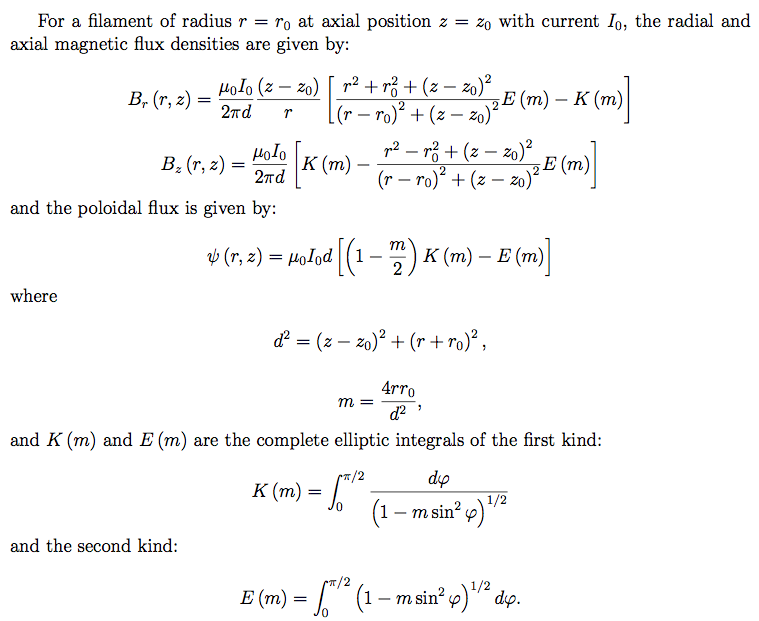
Appendices 8

# Introduction

The HIT-SI lab’s current-equilibrium code (Appendices A and B) in its past state was unorganized, inflexible, and made coil configuration, simulation, and optimization for future machine design, as well as other plasma community projects, inefficient. Because simulation and optimization are required to meet both sufficient machine boundary conditions and minimized coil power/copper supplies (costs), impetus existed to restructure the current-equilibrium code using Python’s object-oriented capabilities. A bottom-up strategy for developing the restructured current-equilibrium code first required the construction of a coil object, Coil() (Appendix C), and subsequently the construction of a coil set object, CoilSet() (Appendix D). By pairing this coil set object with an appropriate plasma coil object, PlasmaCoil() (Appendix F)—designed to approximate the machine’s plasma fields as its own set of coils—a coil simulation object, CoilSim() (Appendix E), was constructed. In addition, a script for calculating proper coil and flux loop (floop) placement was written in order to write to a Google spreadsheet data called by CoilSim(). In the end, this coil simulation object’s functionality was virtually identical to the original current-equilibrium code, yet because of its object-oriented structure, its flexibility was now sufficient for future simulation and optimization.

**Construction of Coil Object and Equations for Coil Fields**

Most of the constructed Coil object’s attributes consisted of the physical parameters given by the following section, drafted by Dr. Brian Nelson:



A complete list of the Coil object’s attributes is documented in Table 1. In addition to these attributes, the method calc\_fields() was constructed within the Coil object, which when called uses the equations above to calculate for a given mesh by breaking each coil into a grid of filaments with appropriate r and z dimensions the radial and axial magnetic fields produced by the coil as well as its magnetic flux, returning these values as well as the location of the filaments and the fields calculation points.   
  
**Table 1, Coil**

|  |
| --- |
| **Attributes Description**  i\_coil:           total current in coil  r\_coil :          radial position of coil  z\_coil :         axial position of coil  r\_width:          radial extent of coil  z\_width:         axial extent of coil  n\_r\_filaments:   number of filaments to approximate coil radially  n\_z\_filaments:   number of filaments to approximate coil axially  r\_floop:         radial flux loop  z\_floop:         axial flux loop  **Method:** calc\_fields() **Description** r\_arr: radial field calculation points z\_arr: axial field calculation points psi: magnetic flux b\_r: radial magnetic flux density  b\_z: axial magnetic flux density r\_filaments: radial locations of filaments z\_filaments: axial locations of filaments |

**Construction of Coil Set Object**

In turn, a CoilSet object designed to model a set of Coil objects was constructed (Appendix E). Its attributes are listed in Table 2. The method, add\_Coil () was written into the CoilSet object, which when called fills the CoilSet object with a supplied, single Coil object. Additionally, a read\_cvs() method was constructed by my coding partner Nan Jiang in order to allow input from a comma-delimited spreadsheet to define a CoilSet. The method set\_mesh() was also defined in order to set an appropriate mesh which can be passed via the final method calc\_fields() to each coil in the CoilSet object, calling their respective calc\_fields() methods in order to produce a matrix of coil fields and locations.

**Table 2, CoilSet**

|  |
| --- |
| **Attributes Description:** coils: list of Coil objects in CoilSet object n\_coils: number of Coil objects in CoilSet object  **Method:** set\_mesh() **Description:**  r\_range: if length is 2 elements, it is assumed that  z\_range: r\_range and z\_range are each the desired min, r\_dim: max points for r and z, with dimensions z\_dim: r\_dim and z\_dim do\_b\_r: Boolean that if True calculates radial B field do\_b\_z: Boolean that if True calculates axial B field  **Method:** calc\_fields() **Description:**  psi: coil set’s flux values  b\_r: coil set’s radial B field values b\_z: coil set’s axial B field values  **Method:** add\_Coil() **Description:**   takes a coil as a parameter and adds it to the   coil set  **Method:** read\_cvs() **Description:**   reads in a Google spreadsheet of coils and their   parameters and adds them to the coil set |

**Construction of Coil Simulation Object**Finally, an object to simulate the fields of the coils and the machine’s plasma (as represented by the plasma coil object discussed below), with functionality virtually identical to the original current-equilibrium code, was constructed (Appendix E). Its attributes are listed in Table 3. The method calc\_fields() was constructed in order to calculate the cumulative fields of the CoilSet, each weighted by their current value, as well as i\_floops and i\_fact, which represent the currents needed to satisfy the machine’s desired equilibrium.

**Table 3, CoilSim**

|  |
| --- |
| **Attributes:** **Description:** coil\_set the CoilSet object in CoilSim plasma\_coil the PlasmaCoil object in CoilSim  **Method:** set\_mesh() **Description:** \*same as CoilSet\* \*same as CoilSet\*  **Method:** calc\_psi() **Description:** psi flux of Coils weighted by current and summed i\_floops currents of flux loops i\_fact same normalized to plasma’s current: 3.2Ma  **Method:** plot() **Description:**  plots all flux contours and machine outline  prints total coil power and current per turn |

**Construction of Plasma Coil Object**

Also, a PlasmaCoil object was constructed (Appendix F) in order to model and plot the machine’s plasma. Its attributes are listed in table 4. The methods set\_mesh() and calc\_psi() were written in the spirit of the CoilSet’s methods of the same name, and a separate method plot\_plasma(), to be called by the CoilSim object to plot the machine’s plasma’s flux on the same graph as the coil set’s, was also written into the PlasmaCoil object.

**Table 4, PlasmaCoil**

|  |
| --- |
| **Attributes:** **Description:** \*same as Coil but with \_p\_\* rzt variables containing plotting data for plasma  tris pt beta **Method:** set\_mesh() **Description:** \*same as CoilSet\* \*same as CoilSet\*  **Method:** calc\_psi() **Description:** \*same as CoilSet\* \*same as CoilSet\*  **Method:** plot\_plasma() **Description:**  plots plasma fields |

**Writing of the Coil/Floop Placement Script**

In addition, a coil/floop placement script was written (Appendix G) with the help of Nan Jiang, which calculates multiple coils’ parameters, largely based on the machine’s geometry, and adds them to a Google spreadsheet. This script drew heavily from the beginning section of the original coil equilibrium code. Some unused sections were removed, however.

**Construction of a Coil GUI**

Upon completion of a CoilSim object, whose functionality mirrored the original coil equilibrium code, a TkInter GUI (Appendix I) was constructed using the CoilSet framework, which provided an intuitive, visual tool for coil parameter optimization. It allowed for coils with varying parameters to be plotted on a chosen mesh along with dynamic field contours, default or filled.

**Further Work**

Future effort will be put towards adding more functionality to the Coil GUI –possibly allowing for machine geometries to be displayed. Also, coil placement optimization software, to be used to aid the design process of future HIT machines, can now proceed given the CoilSet framework and Coil GUI have been completed.

**Appendices**

Appendix A: Original non object-oriented coil field simulation code

hit-pop1m\_coils\_outer\_injector\_more\_coils.py

Appendix B: Accompanying coil field calculating code—multi-make psi

mmpsi2.py

Appendix C: Coil object code  
  
coil\_object.py

Appendix D: Coil set object code

coil\_set\_object.py

Appendix E: Coil Simulation object code

coil\_sim\_object.py

Appendix F: Plasma Coil Object code

plasma\_coil\_object.py

Appendix G: coil/floop placement script  
  
floop\_coil\_placement.py

Appendix H: test script for CoilSim

test.py

Appendix I:

GUI.py

## Appendix A

# -\*- coding: utf-8 -\*-

"""

Created on Thu Apr 19 08:48:12 2012

@author: nelson

"""

from scipy.constants import pi

import pylab as P

from mmpsi import mmpsi

from t3dinp import t3dinp

# dtype definitions for arrays (change here will affect all)

fdtype = P.float64

idtype = P.int16

# resistivity of Cu (Ohm-m)

eta\_Cu = 16.8e-9

# minor radius

a = 1.0

# major radius

r\_0 = 1.5

# radius of vacuum vessel (from r\_0), plus blanket if present

blanket\_width = 0.0

r\_vv0 = a + blanket\_width

# radius of flux conserver to Uberplate feature

a\_u = 0.242

# width of Uberplate

w\_u = 0.35

# width of injector

w\_i = 0.28

# r location of Uberplate (injector feature is perpendicular

# to Uberplate and tangent to the FC)

r\_u = r\_0 + P.sqrt((2 \* a - w\_u) \* a\_u - w\_u \*\* 2 / 4 + a \*\* 2)

# Alternative:

# Could also specific height of Uberplate, z\_u, and calculate a\_u as:

#z\_u = 1.95

#a\_u = (w\_u\*\*2 / 4 + z\_u\*\*2 - a\*\*2) / (2 \* a - w\_u)

# angle of mating point of FC and Uberplate feature

theta\_u = P.arctan2(0.5 \* w\_u + a\_u, a + a\_u)

# center of upper FC to Uberplate feature arc

r\_u\_1 = r\_u

z\_u\_1 = w\_u / 2 + a\_u

# center of lower FC to Uberplate feature arc

r\_u\_2 = r\_u

z\_u\_2 = -(w\_u / 2 + a\_u)

# big coil parameters

b\_coil\_width = 0.1

b\_coils\_list = []

r\_b\_coils\_list = []

z\_b\_coils\_list = []

b\_coils\_r\_width\_list = []

b\_coils\_z\_width\_list = []

r\_floop\_list = []

z\_floop\_list = []

array\_radius = 1.15

array\_r\_width = 0.1

array\_z\_width = 0.1

delta\_theta\_degrees = 15.0

delta\_theta = P.deg2rad(delta\_theta\_degrees)

n\_coils\_each\_side = 5

theta\_start\_degrees = 270.0

theta\_start = P.deg2rad(theta\_start\_degrees)

theta\_end = theta\_start + (n\_coils\_each\_side - 1) \* delta\_theta

for theta in P.linspace(theta\_start, theta\_end, n\_coils\_each\_side):

r\_coil = r\_0 + array\_radius \* P.cos(theta)

z\_coil = array\_radius \* P.sin(theta)

r\_b\_coils\_list.append(r\_coil)

z\_b\_coils\_list.append(z\_coil)

r\_floop = r\_0 + a \* P.cos(theta)

z\_floop = a \* P.sin(theta)

r\_floop\_list.append(r\_floop)

z\_floop\_list.append(z\_floop)

theta\_start\_degrees = 90.0 - (n\_coils\_each\_side - 1) \* delta\_theta\_degrees

theta\_start = P.deg2rad(theta\_start\_degrees)

theta\_end = P.deg2rad(90.0)

for theta in P.linspace(theta\_start, theta\_end, n\_coils\_each\_side):

r\_coil = r\_0 + array\_radius \* P.cos(theta)

z\_coil = array\_radius \* P.sin(theta)

r\_b\_coils\_list.append(r\_coil)

z\_b\_coils\_list.append(z\_coil)

r\_floop = r\_0 + a \* P.cos(theta)

z\_floop = a \* P.sin(theta)

r\_floop\_list.append(r\_floop)

z\_floop\_list.append(z\_floop)

# coils on cryoports

r\_b\_coils\_list.extend([1.16 + 0.5 \* b\_coil\_width, 1.16 + 0.5 \* b\_coil\_width,

0.55 + b\_coil\_width, 0.55 + b\_coil\_width])

z\_b\_coils\_list.extend([-1.1, 1.1, -1.1, 1.1])

# keep index for double-wide coils

i\_wide\_coil = len(z\_b\_coils\_list) - 2

# RKT -> Change coils on injector here

#r\_inj\_coils = [2.9] \* 2

#r\_inj\_coils = [2.8, 3.0, 3.2] \* 2

#r\_inj\_coils = [2.9, 3.2] \* 2

r\_inj\_coils = [2.9, 3.2, 3.5] \* 2

n\_inj\_coils = len(r\_inj\_coils)

z\_inj\_coil = 0.25

z\_inj\_coils = P.ones(n\_inj\_coils, dtype=fdtype) \* z\_inj\_coil

for ii in xrange(int(n\_inj\_coils / 2)):

z\_inj\_coils[ii] \*= -1.0

r\_b\_coils\_list.extend(r\_inj\_coils)

z\_b\_coils\_list.extend(z\_inj\_coils)

# coil at end of injector

inj\_end\_coil = False

if inj\_end\_coil:

r\_b\_coils\_list.append(1.5 + 1.17 + 0.67 + b\_coil\_width)

z\_b\_coils\_list.append(0.0)

# make arrays out of lists

r\_b\_coils = P.array(r\_b\_coils\_list, dtype=fdtype)

z\_b\_upper = r\_vv0 + 0.5 \* b\_coil\_width

z\_b\_inj\_delta = 0.1

z\_b\_coils = P.array(z\_b\_coils\_list, dtype=fdtype)

# total number of big coils

n\_b\_coils = len(r\_b\_coils)

r\_b\_widths = P.ones(n\_b\_coils, dtype=fdtype) \* b\_coil\_width

z\_b\_widths = 1.0 \* r\_b\_widths

n\_r\_b\_elements = P.ones(n\_b\_coils, dtype=idtype) \* 10

n\_z\_b\_elements = 1 \* n\_r\_b\_elements

# double-wide inner coils

r\_b\_widths[i\_wide\_coil:i\_wide\_coil + 2] \*= 2.0

n\_r\_b\_elements[i\_wide\_coil:i\_wide\_coil + 2] \*= 2

#n\_r\_b\_elements[-3:-1] \*= 2

# list entry of small coil positions

r\_s\_width = 0.2

z\_s\_width = r\_s\_width

# RKT -> Change "s" coils here

r\_mod = 0.05 \* 0

z\_mod = 0.05 \* 0

r\_s\_coils = P.array([r\_u - r\_mod, r\_u - r\_mod], dtype=fdtype)

z\_s\_coils = P.array([0.5 \* w\_u + a\_u - z\_mod, -(0.5 \* w\_u + a\_u - z\_mod)],

dtype=fdtype)

# total number of small coils

n\_s\_coils = len(r\_s\_coils)

r\_s\_widths = P.ones(n\_s\_coils, dtype=fdtype) \* r\_s\_width

z\_s\_widths = 1.0 \* r\_s\_widths

n\_r\_s\_elements = P.ones(n\_s\_coils, dtype=idtype) \* 5

n\_z\_s\_elements = 1 \* n\_r\_s\_elements

# plasma coils

i\_p\_coils = P.loadtxt('hitpops.05.txt', delimiter=',', dtype=fdtype)

r\_p\_coils\_full = i\_p\_coils[:, 0]

z\_p\_coils\_full = i\_p\_coils[:, 1]

beta = i\_p\_coils[:, 3] \* 6.28e7

i\_p\_coils\_full = i\_p\_coils[:, 2]

# choose subset where current is not zero

sub = P.where(i\_p\_coils\_full != 0.0)

r\_p\_coils = r\_p\_coils\_full[sub]

z\_p\_coils = z\_p\_coils\_full[sub]

i\_p\_coils = i\_p\_coils\_full[sub]

n\_p\_coils = len(r\_p\_coils)

r\_p\_widths = P.ones(n\_p\_coils, dtype=fdtype) \* 0.05

z\_p\_widths = 1.0 \* r\_p\_widths

n\_r\_p\_elements = P.ones(n\_p\_coils, dtype=idtype)

n\_z\_p\_elements = 1 \* n\_r\_p\_elements

# coil resistances

R\_b\_coils = eta\_Cu \* 2 \* pi \* r\_b\_coils / r\_b\_widths / z\_b\_widths

R\_s\_coils = eta\_Cu \* 2 \* pi \* r\_s\_coils / r\_s\_widths / z\_s\_widths

# fraction of Cu in cross-section

Cu\_fract = 0.9

R\_b\_coils /= Cu\_fract

R\_s\_coils /= Cu\_fract

# flux loops

# common factor for big coil flux loops

fl\_b\_fact = 1.0 / P.sqrt((r\_b\_coils - r\_0) \*\* 2 + z\_b\_coils \*\* 2)

# plasma filaments act as one coil

n\_coils = n\_b\_coils + n\_s\_coils + 1

r\_floops = P.zeros(n\_coils, dtype=fdtype)

z\_floops = 0.0 \* r\_floops

## big coil flux loops

# all on FC on line from (r\_0, 0) to center of coil

r\_floops[0:n\_b\_coils] = r\_0 + a \* (r\_b\_coils - r\_0) \* fl\_b\_fact

z\_floops[0:n\_b\_coils] = a \* z\_b\_coils \* fl\_b\_fact

l\_inj = 0.67 - 0.5 \* w\_i

# put last flux loop at end of injector

if inj\_end\_coil:

for ii in xrange(n\_b\_coils - n\_inj\_coils - 1, n\_b\_coils - 1):

r\_floops[ii] = r\_b\_coils[ii]

z\_floops[ii] = z\_b\_coils[ii] / abs(z\_b\_coils[ii]) \* 0.5 \* w\_i

r\_floops[n\_b\_coils - 1] = r\_u + l\_inj + 0.5 \* w\_i

else:

for ii in xrange(n\_b\_coils - n\_inj\_coils, n\_b\_coils):

r\_floops[ii] = r\_b\_coils[ii]

z\_floops[ii] = z\_b\_coils[ii] / abs(z\_b\_coils[ii]) \* 0.5 \* w\_i

# small coil flux floops

# first two are 45 degrees into the arc transition to injectors

r\_floops[n\_b\_coils] = r\_u\_1 + a\_u \* P.cos(1.25 \* pi)

z\_floops[n\_b\_coils] = z\_u\_1 + a\_u \* P.sin(1.25 \* pi)

r\_floops[n\_b\_coils + 1] = r\_u\_2 + a\_u \* P.cos(0.75 \* pi)

z\_floops[n\_b\_coils + 1] = z\_u\_2 + a\_u \* P.sin(0.75 \* pi)

# plasma current flux loop

r\_floops[-1] = r\_0 - a

z\_floops[-1] = 0.0

n\_floops = len(r\_floops)

# mutual inductance matrix

M = P.zeros([n\_floops, n\_floops], dtype=fdtype)

for ii in xrange(n\_b\_coils):

# set up column vector for each coil's current

# get flux from this coil on all flux loops

r\_arr, z\_arr, psi, b\_r, b\_z, r\_out, z\_out = \

mmpsi(r\_coils=[r\_b\_coils[ii]], r\_widths=[r\_b\_widths[ii]],

n\_r\_elements=[n\_r\_b\_elements[ii]],

z\_coils=[z\_b\_coils[ii]], z\_widths=[z\_b\_widths[ii]],

n\_z\_elements=[n\_z\_b\_elements[ii]],

i\_coils=[1.0],

r\_range=r\_floops, z\_range=z\_floops)

# fill in mutual inductance column

M[:, ii] = psi

for ii in xrange(n\_s\_coils):

# set up column vector for each coil's current

# get flux from this coil on all flux loops

r\_arr, z\_arr, psi, b\_r, b\_z, r\_out, z\_out = \

mmpsi(r\_coils=[r\_s\_coils[ii]], r\_widths=[r\_s\_widths[ii]],

n\_r\_elements=[n\_r\_s\_elements[ii]],

z\_coils=[z\_s\_coils[ii]], z\_widths=[z\_s\_widths[ii]],

n\_z\_elements=[n\_z\_s\_elements[ii]],

i\_coils=[1.0],

r\_range=r\_floops, z\_range=z\_floops)

# fill in mutual inductance column

M[:, n\_b\_coils + ii] = psi

if 'GF\_fl\_p' not in dir():

print "Calculating plasma GF for flux loops, standby ..."

r\_arr, z\_arr, psi, b\_r, b\_z, r\_out, z\_out = \

mmpsi(r\_coils=r\_p\_coils, r\_widths=r\_p\_widths,

n\_r\_elements=n\_r\_p\_elements,

z\_coils=z\_p\_coils, z\_widths=z\_p\_widths,

n\_z\_elements=n\_z\_p\_elements,

i\_coils=i\_p\_coils,

r\_range=r\_floops, z\_range=z\_floops)

print "Done"

# fill in last column of mutual inductance

GF\_fl\_p = psi

M[:, -1] = GF\_fl\_p

# solve for coil currents to make 1 Wb on all flux loops

i\_floops = P.linalg.solve(M, P.ones(n\_floops, dtype=fdtype))

# !!! fix i\_floops here for if any coils are in series !!!

r\_range = P.array([1e-6, 4.0], dtype=fdtype)

z\_range = P.array([-1.5, 1.5], dtype=fdtype)

r\_dim, z\_dim = 100, 100

# Green's function matrix for mesh

GFs = P.zeros([n\_floops, r\_dim, z\_dim])

if 'GFs\_b\_coils' not in dir():

print "Calculating b\_coils GF for mesh, standby ..."

GFs\_b\_coils = P.zeros([n\_b\_coils, r\_dim, z\_dim])

for ii in xrange(n\_b\_coils):

r\_arr, z\_arr, psi, b\_r, b\_z, r\_out, z\_out = \

mmpsi(r\_coils=[r\_b\_coils[ii]], r\_widths=[r\_b\_widths[ii]],

n\_r\_elements=[n\_r\_b\_elements[ii]],

z\_coils=[z\_b\_coils[ii]], z\_widths=[z\_b\_widths[ii]],

n\_z\_elements=[n\_z\_b\_elements[ii]],

i\_coils=[1.0],

r\_range=r\_range, z\_range=z\_range,

r\_dim=r\_dim, z\_dim=z\_dim)

GFs\_b\_coils[ii, ...] = psi

print "Done"

if 'GFs\_s\_coils' not in dir():

print "Calculating s\_coils GF for mesh, standby ..."

GFs\_s\_coils = P.zeros([n\_s\_coils, r\_dim, z\_dim])

for ii in xrange(n\_s\_coils):

r\_arr, z\_arr, psi, b\_r, b\_z, r\_out, z\_out = \

mmpsi(r\_coils=[r\_s\_coils[ii]], r\_widths=[r\_s\_widths[ii]],

n\_r\_elements=[n\_r\_s\_elements[ii]],

z\_coils=[z\_s\_coils[ii]], z\_widths=[z\_s\_widths[ii]],

n\_z\_elements=[n\_z\_s\_elements[ii]],

i\_coils=[1.0],

r\_range=r\_range, z\_range=z\_range,

r\_dim=r\_dim, z\_dim=z\_dim)

GFs\_s\_coils[ii, ...] = psi

print "Done"

if 'GF\_p\_coil' not in dir():

print "Calculating plasma GF for mesh, standby ..."

r\_arr, z\_arr, psi, b\_r, b\_z, r\_out, z\_out = \

mmpsi(r\_coils=r\_p\_coils, r\_widths=r\_p\_widths,

n\_r\_elements=n\_r\_p\_elements,

z\_coils=z\_p\_coils, z\_widths=z\_p\_widths,

n\_z\_elements=n\_z\_p\_elements,

i\_coils=i\_p\_coils,

r\_range=r\_range, z\_range=z\_range, r\_dim=r\_dim, z\_dim=z\_dim)

print "Done"

GF\_p\_coil = psi

GFs[0:n\_b\_coils, ...] = GFs\_b\_coils

GFs[n\_b\_coils:n\_b\_coils + n\_s\_coils, ...] = GFs\_s\_coils

GFs[-1, ...] = GF\_p\_coil

# normalize to I\_p at 3.2 MA

i\_fact = 3.2e6 / i\_floops[-1]

i\_floops \*= i\_fact

# calculate flux on mesh

psi = 0.0

for ii, i\_val in enumerate(i\_floops):

psi += i\_val \* GFs[ii, ...]

# meshes for contour plots

r\_arr, z\_arr = P.meshgrid(P.linspace(r\_range[0], r\_range[1], r\_dim),

P.linspace(z\_range[0], z\_range[1], z\_dim))

# lists of all coils for pcoils function

r\_coils = r\_b\_coils.tolist()

r\_coils.extend(r\_s\_coils.tolist())

z\_coils = z\_b\_coils.tolist()

z\_coils.extend(z\_s\_coils.tolist())

r\_widths = r\_b\_widths.tolist()

r\_widths.extend(r\_s\_widths.tolist())

z\_widths = z\_b\_widths.tolist()

z\_widths.extend(z\_s\_widths.tolist())

def pcoils(r\_coils=r\_coils, r\_widths=r\_widths,

z\_coils=z\_coils, z\_widths=z\_widths, lw=2):

# plots rectangular coils

r\_sq = P.array([0.5, -0.5, -0.5, 0.5, 0.5])

z\_sq = P.array([0.5, 0.5, -0.5, -0.5, 0.5])

for r\_coil, r\_width, z\_coil, z\_width in zip(r\_coils, r\_widths,

z\_coils, z\_widths):

r = r\_coil + r\_sq \* r\_width

z = z\_coil + z\_sq \* z\_width

P.plot(r, z, color='b', lw=lw)

# optional to plot coil on left side

# if plot\_left\_side:

# P.plot(-r, z, color='b')

# FC parameters

# inner midplane gap half-angle

delta\_deg = 2

delta = delta\_deg \* pi / 180.0

# number of points for arcs, etc.

npts = 101

# FC arc from midplane gap to outside injector feature

theta\_1 = P.linspace(pi + delta, 2 \* pi - theta\_u, npts)

theta\_2 = P.linspace(theta\_u, pi - delta, npts)

# outline of the FC, in two parts

r\_FC\_1 = r\_0 + a \* P.cos(theta\_1)

z\_FC\_1 = a \* P.sin(theta\_1)

r\_FC\_2 = r\_0 + a \* P.cos(theta\_2)

z\_FC\_2 = a \* P.sin(theta\_2)

# injector feature angles

theta\_inj\_1 = P.linspace(pi + theta\_u, 1.5 \* pi, npts)

theta\_inj\_2 = P.linspace(0.5 \* pi, pi - theta\_u, npts)

# outline of the injector feature, in two parts

r\_inj\_1 = r\_u\_1 + a\_u \* P.cos(theta\_inj\_1)

z\_inj\_1 = z\_u\_1 + a\_u \* P.sin(theta\_inj\_1)

r\_inj\_2 = r\_u\_2 + a\_u \* P.cos(theta\_inj\_2)

z\_inj\_2 = z\_u\_2 + a\_u \* P.sin(theta\_inj\_2)

#smaller curve feature from Uberplate to injector

r\_tran = z\_u\_1 - a\_u - 0.5 \* w\_i

theta\_tran\_1 = P.linspace(pi, 1.5 \* pi, npts)

r\_tran\_1 = r\_u + r\_tran + r\_tran \* P.cos(theta\_tran\_1)

z\_tran\_1 = 0.5 \* w\_i + r\_tran + r\_tran \* P.sin(theta\_tran\_1)

# default linewidth

lw = 2

# flux conserver color

FC\_col = 'r'

P.clf()

P.axes(aspect='equal')

# plot FC sections and injector transition

P.plot(r\_FC\_1, z\_FC\_1, r\_FC\_2, z\_FC\_2, color=FC\_col, lw=lw)

#P.plot(-r\_FC\_1, z\_FC\_1, -r\_FC\_2, z\_FC\_2, color=FC\_col, lw=lw)

#P.plot(r\_inj\_1, z\_inj\_1, -r\_inj\_1, z\_inj\_1, color=FC\_col, lw=lw)

#P.plot(r\_inj\_2, z\_inj\_2, -r\_inj\_2, z\_inj\_2, color=FC\_col, lw=lw)

P.plot(r\_inj\_1, z\_inj\_1, color=FC\_col, lw=lw)

P.plot(r\_inj\_2, z\_inj\_2, color=FC\_col, lw=lw)

# injector to Uberplate transition feature

P.plot(r\_tran\_1, z\_tran\_1, color=FC\_col, lw=lw)

P.plot(r\_tran\_1, -z\_tran\_1, color=FC\_col, lw=lw)

# sides of injector

#P.plot([r\_u, r\_u + l\_inj],

# [0.5 \* w\_u, 0.5 \* w\_u], color=FC\_col, lw=lw)

#P.plot([r\_u, r\_u + l\_inj],

# [-0.5 \* w\_u, -0.5 \* w\_u], color=FC\_col, lw=lw)

P.plot([r\_u + r\_tran, r\_u + l\_inj],

[0.5 \* w\_i, 0.5 \* w\_i], color=FC\_col, lw=lw)

P.plot([r\_u + r\_tran, r\_u + l\_inj],

[-0.5 \* w\_i, -0.5 \* w\_i], color=FC\_col, lw=lw)

#P.plot([-(r\_0 + 0.5 \* w\_u), -(r\_0 + 0.5 \* w\_u)],

# [z\_u, z\_u + 0.4], color=FC\_col, lw=lw)

#P.plot([-(r\_0 - 0.5 \* w\_u), -(r\_0 - 0.5 \* w\_u)],

# [z\_u, z\_u + 0.4], color=FC\_col, lw=lw)

# end of injector

th = P.linspace(-0.5 \* pi, 0.5 \* pi, npts)

#rs = r\_u + l\_inj + 0.5 \* w\_u \* P.cos(th)

rs = r\_u + l\_inj + 0.5 \* w\_i \* P.cos(th)

#zs = 0.5 \* w\_u \* P.sin(th)

zs = 0.5 \* w\_i \* P.sin(th)

P.plot(rs, zs, color=FC\_col, lw=lw)

#P.plot(-rs, zs, color=FC\_col, lw=lw)

delta2\_deg = 0.0

delta2 = delta2\_deg \* pi / 180.0

r\_bs = r\_u + l\_inj

a\_bs = r\_b\_coils[-1] - 0.5 \* r\_b\_widths[-1] - (r\_u + l\_inj)

a\_b = 0.7

#r\_m = 0.5 / r\_bs \* (a\_b \*\* 2 - a\_bs \*\* 2 + (r\_bs + r\_0) \*\* 2 - r\_0 \*\* 2)

#z\_m = P.sqrt(a\_b \*\* 2 - (r\_m - r\_0) \*\* 2)

#

#theta\_m = P.arctan2(z\_m, r\_m)

#thm = P.linspace(-theta\_m, theta\_m, npts)

# shield for midplane coil

#delta\_m = 17.4 \* pi / 180.0

delta\_m = -5.0 \* pi / 180.0

thm = P.linspace(-0.5 \* pi + delta\_m, 0.5 \* pi - delta\_m, npts)

#P.plot(r\_bs + a\_bs \* P.cos(thm), a\_bs \* P.sin(thm), color='k', lw=lw)

# angle to meet shield at midplane coil

#delta\_b = 6.8 \* pi / 180.0

delta\_b = 11.5 \* pi / 180.0

theta2 = P.linspace(-0.5 \* pi, -delta\_b, npts)

r\_vv = r\_0 + r\_vv0 \* P.cos(theta2)

z\_vv = r\_vv0 \* P.sin(theta2)

#P.plot(r\_vv, z\_vv, color='k', lw=lw)

theta2 = P.linspace(delta\_b, 0.5 \* pi, npts)

r\_vv = r\_0 + r\_vv0 \* P.cos(theta2)

z\_vv = r\_vv0 \* P.sin(theta2)

#P.plot(r\_vv, z\_vv, color='k', lw=lw)

#P.plot(-r\_vv, z\_vv, color='k', lw=lw)

#P.plot([0, r\_vv[-1]], [z\_vv[-1], z\_vv[-1]], color='k', lw=lw)

#P.plot([0, r\_vv[-1]], [-z\_vv[-1], -z\_vv[-1]], color='k', lw=lw)

#tripcolor(r\_pr, z\_pr, p\_pr)

#levels = linspace(psi\_mod.min(), psi\_mod.max(), 30)

#levels = P.arange(-15.0, 15.0 + 0.5, 0.5) \* i\_fact

#delta\_psi = 10.0

#psi\_min = -20.0

#psi\_max = 100.0

#levels = arange(psi\_min, psi\_max + delta\_psi, delta\_psi)

#P.contour(r\_arr, z\_arr, psi, levels=levels)

#P.prism()

#P.hsv()

#P.contour(-r\_arr, z\_arr, psi, levels=levels)

##P.prism()

#P.hsv()

delta\_psi = 0.5

psi\_min = -15.0

psi\_max = 30.0

levels = P.arange(1.0, psi\_max + delta\_psi, delta\_psi) \* i\_fact

P.contour(r\_arr, z\_arr, psi, levels=levels, colors='w')

levels = P.arange(psi\_min, 1.0 + delta\_psi, delta\_psi) \* i\_fact

P.matplotlib.rcParams['contour.negative\_linestyle'] = 'solid'

P.contour(r\_arr, z\_arr, psi, levels=levels, colors='gray')

# need to plot black contours on midplane coil

#rsub = P.where(r\_arr[0, :] >= 5.0)[0]

#levels = P.arange(1.0, psi\_max + delta\_psi, delta\_psi) \* i\_fact

#P.contour(r\_arr[:, rsub], z\_arr[:, rsub], psi[:, rsub],

# levels=levels, colors='gray')

P.plot(r\_floops, z\_floops, 'ro')

#P.plot(-r\_floops, z\_floops, 'ro')

if 'tris' not in dir():

rzt, tris, pt = t3dinp('hitpops.05.t3d')

#do\_conf = False

do\_conf = True

if do\_conf:

P.tricontourf(rzt[:, 0], rzt[:, 1], tris, beta, 1001, zorder=0)

cticks = P.linspace(0.0, 0.2, 5)

P.colorbar(ticks=cticks, format='%.2f')

P.jet()

#no\_text = True

no\_text = False

# whether to annotate in MA or MW

show\_MA = True

if not no\_text:

# annotate coil powers

for ii in xrange(n\_b\_coils):

if i\_floops[ii] >= 0:

signum = '+'

else:

signum = '-'

if show\_MA:

tdata = 1e-6 \* i\_floops[ii]

else:

tdata = 1e-6 \* i\_floops[ii] \*\* 2 \* R\_b\_coils[ii]

# P.text(r\_b\_coils[ii], z\_b\_coils[ii], '%s%3.2f' %

# (signum, 1e-6 \* i\_floops[ii] \*\* 2 \* R\_b\_coils[ii]),

# ha='center', va='center', backgroundcolor='w')

P.text(r\_b\_coils[ii], z\_b\_coils[ii], '%s%3.2f' %

(signum, tdata),

ha='center', va='center', backgroundcolor='w')

if i\_floops[-1] >= 0:

signum = '+'

else:

signum = '-'

P.text(r\_0, 0.0, '%s%3.2f MA' % (signum, 1e-6 \* i\_floops[-1]),

va='center', backgroundcolor='w')

#for ii in [0, 2, 4]:

# for ii in [0, 1]:

for ii in xrange(n\_s\_coils):

if i\_floops[n\_b\_coils + ii] >= 0:

signum = '+'

else:

signum = '-'

if show\_MA:

tdata = 1e-6 \* i\_floops[n\_b\_coils + ii]

else:

tdata = 1e-6 \* i\_floops[n\_b\_coils + ii] \*\* 2 \* \

R\_b\_coils[n\_b\_coils + ii]

# P.text(r\_s\_coils[ii], z\_s\_coils[ii],

# '%s%3.2f' % (signum,

# 1e-6 \* i\_floops[n\_b\_coils + ii] \*\* 2 \*

# R\_s\_coils[ii]),

# va='center', ha='center', backgroundcolor='w')

P.text(r\_s\_coils[ii], z\_s\_coils[ii],

'%s%3.2f' % (signum, tdata),

va='center', ha='center', backgroundcolor='w')

#for ii in [1, 3, 5]:

#for ii in [1]:

# if i\_floops[ii] >= 0:

# signum = '+'

# else:

# signum = '-'

# P.text(r\_s\_coils[ii] - 1.5 \* r\_s\_widths[ii], z\_s\_coils[ii],

# '%s%3.2f' % (signum,

# 1e-6 \* i\_floops[n\_b\_coils + ii] \*\* 2 \*

# R\_s\_coils[ii]),

# va='center', ha='right', backgroundcolor='w')

#text(r\_s\_coils[-1], z\_s\_coils[-1] + 1.5 \* z\_s\_widths[-1], '%3.2f' %

# (1e-6 \* R\_s\_coils[-1] \* i\_floops[n\_b\_coils + n\_s\_coils - 1]\*\*2),

# ha='center', va='center', backgroundcolor='w')

pcoils()

P.axvline()

coil\_power = (i\_floops[:n\_b\_coils] \*\* 2 \* R\_b\_coils).sum() + \

(i\_floops[n\_b\_coils:-1] \*\* 2 \* R\_s\_coils).sum()

coil\_current = abs(i\_floops[:-1]).sum()

if not no\_text:

P.title(r'Total Coil Power = %4.2f MW, Current %4.2f MA-turns' %

(1e-6 \* coil\_power, 1e-6 \* coil\_current))

print r'Total Coil Power = %4.2f MW, Current %4.2f MA-turns' % \

## (1e-6 \* coil\_power, 1e-6 \* coil\_current)

## Appendix B

import pylab as P

import filaments as F

def mmpsi(r\_coils=[0.5, 0.5], r\_widths=[0.05, 0.05], n\_r\_filaments=[10, 10],

z\_coils=[-0.25, 0.25], z\_widths=[0.15, 0.15], n\_z\_filaments=[10, 10],

i\_coils=[1., 1.], r\_range=[0., 1], z\_range=[-1, 1],

r\_dim=20, z\_dim=20, do\_b\_r=False, do\_b\_z=False):

"""

'multiple make psi'

calculates magnetic flux, and optionally radial and axial magnetic

flux density for multiple coils at desired locations

coils are input as sequences (array, list, tuple, etc.) with the

following all having the same dimensions:

r\_coils radial location of centroid of coil

z\_coils axial location of centroid of coil

r\_widths radial extent of coil

z\_widths axial extent of coil

n\_r\_filaments number of filaments to approximate coil in radial direction

n\_z\_filaments number of filaments to approximate coil in axial direction

i\_coils total current (Ampere-turns) in each coil

flux (and fields) are calculated on the domain as given:

r\_range if length is is 2 elements, it is assumed that r\_range

z\_range and z\_range are each the desired min, max points for

r\_dim r and z, with dimensions r\_dim and z\_dim

z\_dim

if length of r\_range is other than two, it is assumed desired locations

for the calculation are in r\_range and z\_range, which must have the

same shape (and r\_dim and z\_dim are ignored)

"""

# if r\_range has 2 elements, it is assumed that r\_range and z\_range

# are each the desired min, max points for r and z, with dimensions

# r\_dim and z\_dim

if len(r\_range) == 2:

r\_arr, z\_arr = P.meshgrid(P.linspace(r\_range[0], r\_range[1], r\_dim),

P.linspace(z\_range[0], z\_range[1], z\_dim))

else:

# otherwise, the user is passing in the r and z locations for

# the desired calucation points

r\_arr, z\_arr = r\_range, z\_range

# poloidal flux (will become the same dimensions as r\_arr and z\_arr)

psi = 0.

# define variables for radial and axial field

if do\_b\_r:

b\_r = 0.

if do\_b\_z:

b\_z = 0.

# list for r and z locations for all filaments

r\_filaments = []

z\_filaments = []

# for each coil

for ii, i\_coil in enumerate(i\_coils):

# current in each filament

I\_0 = i\_coil / n\_r\_filaments[ii] / n\_z\_filaments[ii]

# over all radial filaments

for kk in xrange(1, n\_r\_filaments[ii] + 1):

# r location of filament

r = r\_coils[ii] + 0.5 \* r\_widths[ii] \* \

(1. - (2. \* kk - 1) / n\_r\_filaments[ii])

# over all axial filaments

for jj in xrange(1, n\_z\_filaments[ii] + 1):

# z location of filament

z = z\_coils[ii] + 0.5 \* z\_widths[ii] \* \

(1. - (2. \* jj - 1) / n\_z\_filaments[ii])

# add r and z location to filaments

r\_filaments.append(r)

z\_filaments.append(z)

# calculate flux over entire r\_arr and z\_arr for

# filament at (r, z) with current I\_0

psi += F.poloidal\_flux\_filament(r\_arr, z\_arr, r, z, I\_0)

# likewise for B\_R

if do\_b\_r:

b\_r += F.b\_r\_filament(r\_arr, z\_arr, r, z, I\_0)

# and B\_Z

if do\_b\_z:

b\_z += F.b\_z\_filament(r\_arr, z\_arr, r, z, I\_0)

# dummy arrays for B\_R and B\_Z if they were NOT calculated

if not do\_b\_r:

b\_r = 0. \* psi

if not do\_b\_z:

b\_z = 0. \* psi

# convert lists to NumPy arrays

r\_filaments, z\_filaments = P.array(r\_filaments), P.array(z\_filaments)

# return calculation domain, flux, B\_R, B\_Z, and filament locations

return r\_arr, z\_arr, psi, b\_r, b\_z, r\_filaments, z\_filaments

**Appendix C**

# -\*- coding: utf-8 -\*-

"""

Created on Tue Feb 4 13:57:15 2014

@author: zchmlk

"""

import pylab as P

import filaments as F

class Coil(object):

"""

magnetic field Coil object

set \_\_init\_\_() method for inputs

Methods:

calc\_fields() calculates flux (and optionally field) on desired domain

"""

def \_\_init\_\_(self, i\_coil=1.0, r\_width=0.1, z\_width=0.1, r\_coil=0.5,

z\_coil=0.25, n\_r\_filaments=6, n\_z\_filaments=6,

r\_floop=1, z\_floop=1):

"""

Inputs:

i\_coil total current in coil [Ampere-turns]

r\_coil radial position of coil [m]

z\_coil axial position of coil [m]

r\_width radial extent of coil [m]

z\_width axial extent of coil [m]

n\_r\_filaments number of filaments to approximate coil radially

n\_z\_filaments number of filaments to approximate coil axially

r\_floop radial flux loop

z\_floop axial flux loop

"""

self.i\_coil = i\_coil

self.r\_width = r\_width

self.z\_width = z\_width

self.r\_coil = r\_coil

self.z\_coil = z\_coil

self.n\_r\_filaments = n\_r\_filaments

self.n\_z\_filaments = n\_z\_filaments

self.r\_floop = r\_floop

self.z\_floop = z\_floop

"""reference mmpsi2.py for calc\_fields comments"""

def calc\_fields(self, r\_range=[0., 1.], z\_range=[-1., 1.],

r\_dim=20, z\_dim=20, do\_b\_r=False, do\_b\_z=False):

if len(r\_range) == 2:

r\_arr, z\_arr = P.meshgrid(P.linspace(r\_range[0], r\_range[1],

r\_dim),

P.linspace(z\_range[0], z\_range[1],

z\_dim))

else:

r\_arr, z\_arr = r\_range, z\_range

psi = 0.

if do\_b\_r:

b\_r = 0.

if do\_b\_z:

b\_z = 0.

r\_filaments = []

z\_filaments = []

for kk in xrange(1, self.n\_r\_filaments + 1):

r = self.r\_coil + 0.5 \* self.r\_width \* \

(1. - (2. \* kk - 1) / self.n\_r\_filaments)

for jj in xrange(1, self.n\_z\_filaments + 1):

z = self.z\_coil + 0.5 \* self.z\_width \* \

(1. - (2. \* jj - 1) / self.n\_z\_filaments)

r\_filaments.append(r)

z\_filaments.append(z)

psi += F.poloidal\_flux\_filament(r\_arr, z\_arr, r, z,

self.i\_coil /

self.n\_r\_filaments /

self.n\_z\_filaments)

if do\_b\_r:

b\_r += F.b\_r\_filament(r\_arr, z\_arr, r, z, self.i\_coil)

if do\_b\_z:

b\_z += F.b\_z\_filament(r\_arr, z\_arr, r, z, self.i\_coil)

if not do\_b\_r:

b\_r = 0. \* psi

if not do\_b\_z:

b\_z = 0. \* psi

r\_filaments, z\_filaments = P.array(r\_filaments), P.array(z\_filaments)

self.r\_arr = r\_arr

self.z\_arr = z\_arr

self.psi = psi

self.b\_r = b\_r

self.b\_z = b\_z

self.r\_filaments = r\_filaments

self.z\_filaments = z\_filaments

return r\_arr, z\_arr, psi, b\_r, b\_z, r\_filaments, z\_filaments

**Appendix D**

# -\*- coding: utf-8 -\*-

"""

Created on Thu Mar 27 14:24:19 2014

@author: nan

"""

from coil\_object import Coil

#import gspread

import pylab as P

fdtype = P.float64

class CoilSet(object):

"""

set of Coil() objects

.

set \_\_init\_\_() method for inputs

Methods:

calc\_fields() calculates flux (and optionally field)

of set of coils on desired domain

contourf() plots filled contour of flux on domain

"""

def \_\_init\_\_(self, coils=[]):

"""

Inputs:

coils array of Coil() objects

"""

self.coils = coils

self.n\_coils = len(coils)

def add\_Coil(self, coil, r\_floops\_list=[], z\_floops\_list=[],

r\_coils\_list=[], z\_coils\_list=[],

r\_widths\_list=[], z\_widths\_list=[]):

if type(coil) != Coil:

print "Error in add\_Coil method, input not Coil object"

return

self.coils.append(coil)

self.n\_coils = len(self.coils)

r\_floops\_list.append(coil.r\_floop)

z\_floops\_list.append(coil.z\_floop)

r\_coils\_list.append(coil.r\_coil)

z\_coils\_list.append(coil.z\_coil)

r\_widths\_list.append(coil.r\_width)

z\_widths\_list.append(coil.z\_width)

# r\_floops, z\_floops, r\_coils, z\_coils, r\_widths, z\_widths = \

# P.array(r\_floops\_list), P.array(z\_floops\_list), \

# P.array(r\_coils\_list), P.array(z\_coils\_list), \

# P.array(r\_widths\_list), P.array(z\_widths\_list)

self.r\_floops = P.array(r\_floops\_list)

self.z\_floops = P.array(z\_floops\_list)

self.r\_coils = P.array(r\_coils\_list)

self.z\_coils = P.array(z\_coils\_list)

self.r\_widths = P.array(r\_widths\_list)

self.z\_widths = P.array(z\_widths\_list)

def remove\_Coil(self, coil):

if type(coil) != Coil:

print "Error in remove\_Coil method, input not Coil object"

return

coilSetTemp = []

self.n\_coils = 0

for coilSetCoil in self.coils:

if (coilSetCoil.r\_coil != coil.r\_coil) and \

(coilSetCoil.z\_coil != coil.z\_coil):

coilSetTemp.append(coilSetCoil)

self.coils = None

self.coils = []

for coilTemp in coilSetTemp:

self.add\_Coil(coilTemp)

def read\_cvs(self, filename='coil\_set\_spreadsheet',

username='hitsiexperiment@gmail.com', password='2vinjcity'):

'''@author: NJ

before using this method, install gspread in your device

Do not include empty rows in google spread sheet'''

r\_floops\_list = []

z\_floops\_list = []

r\_coils\_list = []

z\_coils\_list = []

r\_widths\_list = []

z\_widths\_list = []

print "Reading in coil parameters, standby..."

# login to a Google account

gc = gspread.login(username, password)

# read in Google spreadsheet

spread\_sheet = gc.open(filename)

worksheet = spread\_sheet.sheet1

values = worksheet.get\_all\_values()

for ii in xrange(1, len(values)):

coil = Coil()

coil.i\_coil = float(values[ii][0])

coil.r\_coil = float(values[ii][1])

coil.z\_coil = float(values[ii][2])

coil.r\_width = float(values[ii][3])

coil.z\_width = float(values[ii][4])

coil.n\_r\_filaments = int(float(values[ii][5]))

coil.n\_z\_filaments = int(float(values[ii][6]))

coil.r\_floop = float(float(values[ii][7]))

coil.z\_floop = float(float(values[ii][8]))

self.coils.append(coil)

r\_floops\_list.append(coil.r\_floop)

z\_floops\_list.append(coil.z\_floop)

r\_coils\_list.append(coil.r\_coil)

z\_coils\_list.append(coil.z\_coil)

r\_widths\_list.append(coil.r\_width)

z\_widths\_list.append(coil.z\_width)

self.n\_coils = len(self.coils)

# r\_floops, z\_floops, r\_coils, z\_coils, r\_widths, z\_widths = \

# P.array(r\_floops\_list), P.array(z\_floops\_list), \

# P.array(r\_coils\_list), P.array(z\_coils\_list), \

# P.array(r\_widths\_list), P.array(z\_widths\_list)

#

# self.r\_floops = r\_floops

# self.z\_floops = z\_floops

# self.r\_coils = r\_coils

# self.z\_coils = z\_coils

# self.r\_widths = r\_widths

# self.z\_widths = z\_widths

self.r\_floops = P.array(r\_floops\_list)

self.z\_floops = P.array(z\_floops\_list)

self.r\_coils = P.array(r\_coils\_list)

self.z\_coils = P.array(z\_coils\_list)

self.r\_widths = P.array(r\_widths\_list)

self.z\_widths = P.array(z\_widths\_list)

def calc\_fields(self, r\_range=[0., 1], z\_range=[-1, 1],

r\_dim=20, z\_dim=20,

do\_b\_r=False, do\_b\_z=False):

self.r\_range = r\_range

self.z\_range = z\_range

self.r\_dim = r\_dim

self.z\_dim = z\_dim

self.do\_b\_r = do\_b\_r

self.do\_b\_z = do\_b\_z

for coil in self.coils:

coil.r\_range = r\_range

coil.z\_range = z\_range

coil.r\_dim = r\_dim

coil.z\_dim = z\_dim

coil.do\_b\_r = do\_b\_r

coil.do\_b\_z = do\_b\_z

self.mesh = r\_range, z\_range, r\_dim, z\_dim, do\_b\_r, do\_b\_z

self.psi = 0.0

self.b\_r = 0.0

self.b\_z = 0.0

psi\_list = []

b\_r\_list = []

b\_z\_list = []

for ii in xrange(self.n\_coils):

r\_arr, z\_arr, psi, b\_r, b\_z, r\_filaments, z\_filaments = \

self.coils[ii].calc\_fields(r\_range=self.r\_range,

z\_range=self.z\_range,

r\_dim=self.r\_dim,

z\_dim=self.z\_dim,

do\_b\_r=self.do\_b\_r,

do\_b\_z=self.do\_b\_z)

psi\_list.append(psi)

b\_r\_list.append(b\_r)

b\_z\_list.append(b\_z)

self.r\_arr = r\_arr

self.z\_arr = z\_arr

psi, b\_r, b\_z = P.array(psi\_list), P.array(b\_r\_list), P.array(b\_z\_list)

self.psi = psi

self.b\_r = b\_r

self.b\_z = b\_z

psi\_cumulative = 0.0

for ii in xrange(self.n\_coils):

psi\_cumulative += self.coils[ii].i\_coil \* self.psi[ii]

self.psi\_cumulative = psi\_cumulative

return psi, b\_r, b\_z

## Appendix E

# -\*- coding: utf-8 -\*-

"""

Created on Sat Apr 12 18:37:00 2014

@author: zchmlk

"""

import pylab as P

from scipy.constants import pi

from coil\_set\_object import CoilSet

from plasma\_coil\_object import PlasmaCoil

fdtype = P.float64

class CoilSim(object):

def \_\_init\_\_(self, coil\_set=CoilSet(), plasma\_coil=PlasmaCoil()):

self.coil\_set = coil\_set

self.plasma\_coil = plasma\_coil

def calc\_psi(self):

# combine coil floops with plasma floop

r\_floops\_total\_list = list(self.coil\_set.r\_floops)

r\_floops\_total\_list.extend([self.plasma\_coil.r\_floop])

z\_floops\_total\_list = list(self.coil\_set.z\_floops)

z\_floops\_total\_list.extend([self.plasma\_coil.z\_floop])

r\_floops\_total = P.array(r\_floops\_total\_list)

z\_floops\_total = P.array(z\_floops\_total\_list)

self.r\_floops\_total = r\_floops\_total

self.z\_floops\_total = z\_floops\_total

self.n\_floops = len(r\_floops\_total)

# calc coils GFs for floops

psi\_coils\_floops, b\_r, b\_z = \

self.coil\_set.calc\_fields(r\_range=self.r\_floops\_total,

z\_range=self.z\_floops\_total)

self.psi\_coils\_floops = psi\_coils\_floops

# calc plasma coil GFs for floops

r\_arr, z\_arr, psi\_plasma\_coil\_floops, b\_r, b\_z, r\_out, z\_out = \

self.plasma\_coil.calc\_fields(r\_range=self.r\_floops\_total,

z\_range=self.z\_floops\_total)

self.psi\_plasma\_coil\_floops = psi\_plasma\_coil\_floops

# combine all GFs for floops

total\_psi\_floops\_list = list(psi\_coils\_floops)

total\_psi\_floops\_list.append(psi\_plasma\_coil\_floops)

total\_psi\_floops = P.array(total\_psi\_floops\_list)

self.total\_psi\_floops = total\_psi\_floops

# make mutual inductance matrix

M = P.zeros([self.n\_floops, self.n\_floops], dtype=fdtype)

# fill up M (NOT RIGHT, /100 AND OFF BY A BIT)

M[:, 0:self.n\_floops] = total\_psi\_floops

# self.M = M

# for some reason we need the transpose of M made this way

self.M = M.T

# solve for coil currents to make 1 Wb on all flux loops

i\_floops = P.linalg.solve(self.M, P.ones(self.n\_floops, dtype=fdtype))

# make GFs

GFs = P.zeros([self.n\_floops, 100, 100])

# calc GFs for coils

psi\_coils, b\_r, b\_z = \

self.coil\_set.calc\_fields(r\_range=P.array([1e-6, 4.0],

dtype=fdtype),

z\_range=P.array([-1.5, 1.5],

dtype=fdtype),

r\_dim=100, z\_dim=100)

self.psi\_coils = psi\_coils

# calc GF for plasma coil

r\_arr, z\_arr, psi\_plasma\_coil, b\_r, b\_z, r\_out, z\_out = \

self.plasma\_coil.calc\_fields(r\_range=P.array([1e-6, 4.0],

dtype=fdtype),

z\_range=P.array([-1.5, 1.5],

dtype=fdtype),

r\_dim=100, z\_dim=100)

self.psi\_plasma\_coil = psi\_plasma\_coil

# fill up GFs

GFs[0:self.coil\_set.n\_coils, ...] = psi\_coils

GFs[-1, ...] = psi\_plasma\_coil

self.GFs = GFs

# normalize to I\_p at 3.2 MA

i\_fact = 3.2e6 / i\_floops[-1]

i\_floops \*= i\_fact

# calculate flux on mesh

psi = 0.0

for ii, i\_val in enumerate(i\_floops):

psi += i\_val \* GFs[ii, ...]

self.i\_floops = i\_floops

self.i\_fact = i\_fact

self.psi = psi

return psi

def plot(self):

self.calc\_psi()

r\_range = P.array([1e-6, 4.0], dtype=fdtype)

z\_range = P.array([-1.5, 1.5], dtype=fdtype)

r\_dim = 100

z\_dim = 100

# resistivity of Cu (Ohm-m)

eta\_Cu = 16.8e-9

# coil resistances

R\_b\_coils = eta\_Cu \* 2 \* pi \* self.coil\_set.r\_coils / \

self.coil\_set.r\_widths / self.coil\_set.z\_widths

# minor radius

a = 1.0

# major radius

r\_0 = 1.5

# radius of vacuum vessel (from r\_0), plus blanket if present

blanket\_width = 0.0

r\_vv0 = a + blanket\_width

# radius of flux conserver to Uberplate feature

a\_u = 0.242

# width of Uberplate

w\_u = 0.35

# width of injector

w\_i = 0.28

# r location of Uberplate (injector feature is perpendicular

# to Uberplate and tangent to the FC)

r\_u = r\_0 + P.sqrt((2 \* a - w\_u) \* a\_u - w\_u \*\* 2 / 4 + a \*\* 2)

# angle of mating point of FC and Uberplate feature

theta\_u = P.arctan2(0.5 \* w\_u + a\_u, a + a\_u)

# center of upper FC to Uberplate feature arc

r\_u\_1 = r\_u

z\_u\_1 = w\_u / 2 + a\_u

# center of lower FC to Uberplate feature arc

r\_u\_2 = r\_u

z\_u\_2 = -(w\_u / 2 + a\_u)

# meshes for contour plots

r\_arr, z\_arr = P.meshgrid(P.linspace(r\_range[0], r\_range[1], r\_dim),

P.linspace(z\_range[0], z\_range[1], z\_dim))

def pcoils(r\_coils=self.coil\_set.r\_coils,

r\_widths=self.coil\_set.r\_widths,

z\_coils=self.coil\_set.z\_coils,

z\_widths=self.coil\_set.z\_widths, lw=2):

# plots rectangular coils

r\_sq = P.array([0.5, -0.5, -0.5, 0.5, 0.5])

z\_sq = P.array([0.5, 0.5, -0.5, -0.5, 0.5])

for r\_coil, r\_width, z\_coil, z\_width in zip(r\_coils, r\_widths,

z\_coils, z\_widths):

r = r\_coil + r\_sq \* r\_width

z = z\_coil + z\_sq \* z\_width

P.plot(r, z, color='b', lw=lw)

# optional to plot coil on left side

# if plot\_left\_side:

# P.plot(-r, z, color='b')

# FC parameters

# inner midplane gap half-angle

delta\_deg = 2

delta = delta\_deg \* pi / 180.0

# number of points for arcs, etc.

npts = 101

# FC arc from midplane gap to outside injector feature

theta\_1 = P.linspace(pi + delta, 2 \* pi - theta\_u, npts)

theta\_2 = P.linspace(theta\_u, pi - delta, npts)

# outline of the FC, in two parts

r\_FC\_1 = r\_0 + a \* P.cos(theta\_1)

z\_FC\_1 = a \* P.sin(theta\_1)

r\_FC\_2 = r\_0 + a \* P.cos(theta\_2)

z\_FC\_2 = a \* P.sin(theta\_2)

# injector feature angles

theta\_inj\_1 = P.linspace(pi + theta\_u, 1.5 \* pi, npts)

theta\_inj\_2 = P.linspace(0.5 \* pi, pi - theta\_u, npts)

# outline of the injector feature, in two parts

r\_inj\_1 = r\_u\_1 + a\_u \* P.cos(theta\_inj\_1)

z\_inj\_1 = z\_u\_1 + a\_u \* P.sin(theta\_inj\_1)

r\_inj\_2 = r\_u\_2 + a\_u \* P.cos(theta\_inj\_2)

z\_inj\_2 = z\_u\_2 + a\_u \* P.sin(theta\_inj\_2)

l\_inj = 0.67 - 0.5 \* w\_i

#smaller curve feature from Uberplate to injector

r\_tran = z\_u\_1 - a\_u - 0.5 \* w\_i

theta\_tran\_1 = P.linspace(pi, 1.5 \* pi, npts)

r\_tran\_1 = r\_u + r\_tran + r\_tran \* P.cos(theta\_tran\_1)

z\_tran\_1 = 0.5 \* w\_i + r\_tran + r\_tran \* P.sin(theta\_tran\_1)

# default linewidth

lw = 2

# flux conserver color

FC\_col = 'r'

P.clf()

P.axes(aspect='equal')

# plot FC sections and injector transition

P.plot(r\_FC\_1, z\_FC\_1, r\_FC\_2, z\_FC\_2, color=FC\_col, lw=lw)

#P.plot(-r\_FC\_1, z\_FC\_1, -r\_FC\_2, z\_FC\_2, color=FC\_col, lw=lw)

#P.plot(r\_inj\_1, z\_inj\_1, -r\_inj\_1, z\_inj\_1, color=FC\_col, lw=lw)

#P.plot(r\_inj\_2, z\_inj\_2, -r\_inj\_2, z\_inj\_2, color=FC\_col, lw=lw)

P.plot(r\_inj\_1, z\_inj\_1, color=FC\_col, lw=lw)

P.plot(r\_inj\_2, z\_inj\_2, color=FC\_col, lw=lw)

# injector to Uberplate transition feature

P.plot(r\_tran\_1, z\_tran\_1, color=FC\_col, lw=lw)

P.plot(r\_tran\_1, -z\_tran\_1, color=FC\_col, lw=lw)

# sides of injector

#P.plot([r\_u, r\_u + l\_inj],

# [0.5 \* w\_u, 0.5 \* w\_u], color=FC\_col, lw=lw)

#P.plot([r\_u, r\_u + l\_inj],

# [-0.5 \* w\_u, -0.5 \* w\_u], color=FC\_col, lw=lw)

P.plot([r\_u + r\_tran, r\_u + l\_inj],

[0.5 \* w\_i, 0.5 \* w\_i], color=FC\_col, lw=lw)

P.plot([r\_u + r\_tran, r\_u + l\_inj],

[-0.5 \* w\_i, -0.5 \* w\_i], color=FC\_col, lw=lw)

#P.plot([-(r\_0 + 0.5 \* w\_u), -(r\_0 + 0.5 \* w\_u)],

# [z\_u, z\_u + 0.4], color=FC\_col, lw=lw)

#P.plot([-(r\_0 - 0.5 \* w\_u), -(r\_0 - 0.5 \* w\_u)],

# [z\_u, z\_u + 0.4], color=FC\_col, lw=lw)

# end of injector

th = P.linspace(-0.5 \* pi, 0.5 \* pi, npts)

#rs = r\_u + l\_inj + 0.5 \* w\_u \* P.cos(th)

rs = r\_u + l\_inj + 0.5 \* w\_i \* P.cos(th)

#zs = 0.5 \* w\_u \* P.sin(th)

zs = 0.5 \* w\_i \* P.sin(th)

P.plot(rs, zs, color=FC\_col, lw=lw)

#P.plot(-rs, zs, color=FC\_col, lw=lw)

delta2\_deg = 0.0

delta2 = delta2\_deg \* pi / 180.0

r\_bs = r\_u + l\_inj

a\_bs = self.coil\_set.r\_coils[-3] - 0.5 \* \

self.coil\_set.r\_widths[-3] - (r\_u + l\_inj)

a\_b = 0.7

#r\_m = 0.5 / r\_bs \* (a\_b \*\* 2 - a\_bs \*\* 2 + \

# (r\_bs + r\_0) \*\* 2 - r\_0 \*\* 2)

#z\_m = P.sqrt(a\_b \*\* 2 - (r\_m - r\_0) \*\* 2)

#

#theta\_m = P.arctan2(z\_m, r\_m)

#thm = P.linspace(-theta\_m, theta\_m, npts)

# shield for midplane coil

#delta\_m = 17.4 \* pi / 180.0

delta\_m = -5.0 \* pi / 180.0

thm = P.linspace(-0.5 \* pi + delta\_m, 0.5 \* pi - delta\_m, npts)

#P.plot(r\_bs + a\_bs \* P.cos(thm), a\_bs \* P.sin(thm), color='k', lw=lw)

# angle to meet shield at midplane coil

#delta\_b = 6.8 \* pi / 180.0

delta\_b = 11.5 \* pi / 180.0

theta2 = P.linspace(-0.5 \* pi, -delta\_b, npts)

r\_vv = r\_0 + r\_vv0 \* P.cos(theta2)

z\_vv = r\_vv0 \* P.sin(theta2)

#P.plot(r\_vv, z\_vv, color='k', lw=lw)

theta2 = P.linspace(delta\_b, 0.5 \* pi, npts)

r\_vv = r\_0 + r\_vv0 \* P.cos(theta2)

z\_vv = r\_vv0 \* P.sin(theta2)

#P.plot(r\_vv, z\_vv, color='k', lw=lw)

#P.plot(-r\_vv, z\_vv, color='k', lw=lw)

#P.plot([0, r\_vv[-1]], [z\_vv[-1], z\_vv[-1]], color='k', lw=lw)

#P.plot([0, r\_vv[-1]], [-z\_vv[-1], -z\_vv[-1]], color='k', lw=lw)

#tripcolor(r\_pr, z\_pr, p\_pr)

#levels = linspace(psi\_mod.min(), psi\_mod.max(), 30)

#levels = P.arange(-15.0, 15.0 + 0.5, 0.5) \* i\_fact

#delta\_psi = 10.0

#psi\_min = -20.0

#psi\_max = 100.0

#levels = arange(psi\_min, psi\_max + delta\_psi, delta\_psi)

#P.contour(r\_arr, z\_arr, psi, levels=levels)

#P.prism()

#P.hsv()

#P.contour(-r\_arr, z\_arr, psi, levels=levels)

##P.prism()

#P.hsv()

delta\_psi = 0.5

psi\_min = -15.0

psi\_max = 30.0

self.plasma\_coil.plot\_plasma()

levels = P.arange(1.0, psi\_max + delta\_psi, delta\_psi) \* self.i\_fact

P.contour(r\_arr, z\_arr, self.psi, levels=levels, colors='w')

levels = P.arange(psi\_min, 1.0 + delta\_psi, delta\_psi) \* self.i\_fact

P.matplotlib.rcParams['contour.negative\_linestyle'] = 'solid'

P.contour(r\_arr, z\_arr, self.psi, levels=levels, colors='gray')

# need to plot black contours on midplane coil

#rsub = P.where(r\_arr[0, :] >= 5.0)[0]

#levels = P.arange(1.0, psi\_max + delta\_psi, delta\_psi) \* i\_fact

#P.contour(r\_arr[:, rsub], z\_arr[:, rsub], psi[:, rsub],

# levels=levels, colors='gray')

P.plot(self.coil\_set.r\_floops, self.coil\_set.z\_floops, 'ro')

#P.plot(-r\_floops, z\_floops, 'ro')

#no\_text = True

no\_text = False

# whether to annotate in MA or MW

show\_MA = True

if not no\_text:

# annotate coil powers

for ii in xrange(self.coil\_set.n\_coils):

if self.i\_floops[ii] >= 0:

signum = '+'

else:

signum = '-'

if show\_MA:

tdata = 1e-6 \* self.i\_floops[ii]

else:

tdata = 1e-6 \* self.i\_floops[ii] \*\* 2 \* R\_b\_coils[ii]

# P.text(r\_b\_coils[ii], z\_b\_coils[ii], '%s%3.2f' %

# (signum, 1e-6 \* i\_floops[ii] \*\* 2 \* R\_b\_coils[ii]),

# ha='center', va='center', backgroundcolor='w')

P.text(self.coil\_set.r\_coils[ii], self.coil\_set.z\_coils[ii],

'%s%3.2f' % (signum, tdata),

ha='center', va='center', backgroundcolor='w')

if self.i\_floops[-1] >= 0:

signum = '+'

else:

signum = '-'

P.text(r\_0, 0.0, '%s%3.2f MA' % (signum, 1e-6 \* self.i\_floops[-1]),

va='center', backgroundcolor='w')

pcoils()

P.axvline()

coil\_power = (self.i\_floops[:self.coil\_set.n\_coils] \*\* 2 \*

R\_b\_coils).sum()

coil\_current = abs(self.i\_floops[:-1]).sum()

if not no\_text:

P.title(r'Total Coil Power = %4.2f MW, Current %4.2f MA-turns' %

(1e-6 \* coil\_power, 1e-6 \* coil\_current))

print r'Total Coil Power = %4.2f MW, Current %4.2f MA-turns' % \

(1e-6 \* coil\_power, 1e-6 \* coil\_current)

self.r\_arr = r\_arr

self.z\_arr = z\_arr

self.levels = levels

## Appendix F

# -\*- coding: utf-8 -\*-

"""

Created on Sun Apr 6 23:50:03 2014

@author: zchmlk

"""

import pylab as P

import filaments as F

from t3dinp import t3dinp

fdtype = P.float64

idtype = P.int16

class PlasmaCoil(object):

def \_\_init\_\_(self, r\_floop=0.5, z\_floop=0.0,

i\_p\_coil\_filename='hitpops.05.txt',

tris\_filename='hitpops.05.t3d'):

self.r\_floop = r\_floop

self.z\_floop = z\_floop

# read equilibrium file

i\_p\_coils = P.loadtxt(i\_p\_coil\_filename, delimiter=',', dtype=fdtype)

self.i\_p\_coils = i\_p\_coils

r\_p\_coils\_full = i\_p\_coils[:, 0]

z\_p\_coils\_full = i\_p\_coils[:, 1]

# ??? what is this scale factor, something to do with mu\_0 ???

beta = i\_p\_coils[:, 3] \* 6.28e7

i\_p\_coils\_full = i\_p\_coils[:, 2]

self.r\_p\_coils\_full = r\_p\_coils\_full

self.z\_p\_coils\_full = z\_p\_coils\_full

self.beta = beta

self.i\_p\_coils\_full = i\_p\_coils\_full

# choose subset where current is not zero

sub = P.where(i\_p\_coils\_full != 0.0)

r\_p\_coils = r\_p\_coils\_full[sub]

z\_p\_coils = z\_p\_coils\_full[sub]

i\_p\_coils = i\_p\_coils\_full[sub]

n\_p\_coils = len(r\_p\_coils)

self.r\_p\_coils = r\_p\_coils

self.z\_p\_coils = z\_p\_coils

self.i\_p\_coils = i\_p\_coils

self.n\_p\_coils = n\_p\_coils

r\_p\_widths = P.ones(n\_p\_coils, dtype=fdtype) \* 0.05

z\_p\_widths = 1.0 \* r\_p\_widths

n\_r\_p\_filaments = P.ones(n\_p\_coils, dtype=idtype)

n\_z\_p\_filaments = 1 \* n\_r\_p\_filaments

self.r\_p\_widths = r\_p\_widths

self.z\_p\_widths = z\_p\_widths

self.n\_r\_p\_filaments = n\_r\_p\_filaments

self.n\_z\_p\_filaments = n\_z\_p\_filaments

# read in triangle unstructured mesh information

rzt, tris, pt = t3dinp(tris\_filename)

self.rzt = rzt

self.tris = tris

self.pt = pt

def calc\_fields(self, r\_range=[0., 1], z\_range=[-1, 1],

r\_dim=20, z\_dim=20, do\_b\_r=False, do\_b\_z=False):

self.r\_range = r\_range

self.z\_range = z\_range

self.r\_dim = r\_dim

self.z\_dim = z\_dim

self.do\_b\_r = do\_b\_r

self.do\_b\_z = do\_b\_z

self.mesh = r\_range, z\_range, r\_dim, z\_dim, do\_b\_r, do\_b\_z

# !!! not sure if 'mesh' will ever appear in this context !!!

if 'mesh' not in dir(self):

self.set\_mesh()

if len(self.r\_range) == 2:

r\_arr, z\_arr = P.meshgrid(P.linspace(self.r\_range[0],

self.r\_range[1],

self.r\_dim),

P.linspace(self.z\_range[0],

self.z\_range[1],

self.z\_dim))

else:

r\_arr, z\_arr = self.r\_range, self.z\_range

psi = 0.

if self.do\_b\_r:

b\_r = 0.

if self.do\_b\_z:

b\_z = 0.

r\_out = []

z\_out = []

for ii, i\_coil in enumerate(self.i\_p\_coils):

I\_0 = i\_coil / self.n\_r\_p\_filaments[ii] / self.n\_z\_p\_filaments[ii]

for kk in xrange(1, self.n\_r\_p\_filaments[ii] + 1):

r = self.r\_p\_coils[ii] + 0.5 \* self.r\_p\_widths[ii] \* \

(1. - (2. \* kk - 1) / self.n\_r\_p\_filaments[ii])

for jj in xrange(1, self.n\_z\_p\_filaments[ii] + 1):

z = self.z\_p\_coils[ii] + 0.5 \* self.z\_p\_widths[ii] \* \

(1. - (2. \* jj - 1) / self.n\_z\_p\_filaments[ii])

r\_out.append(r)

z\_out.append(z)

psi += F.poloidal\_flux\_filament(r\_arr, z\_arr, r, z, I\_0)

if self.do\_b\_r:

b\_r += F.b\_r\_filament(r\_arr, z\_arr, r, z, I\_0)

if self.do\_b\_z:

b\_z += F.b\_z\_filament(r\_arr, z\_arr, r, z, I\_0)

if not self.do\_b\_r:

b\_r = 0. \* psi

if not self.do\_b\_z:

b\_z = 0. \* psi

self.r\_arr = r\_arr

self.z\_arr = z\_arr

self.psi = psi

self.b\_r = b\_r

self.b\_z = b\_z

self.r\_out = r\_out

self.z\_out = z\_out

return r\_arr, z\_arr, psi, b\_r, b\_z, r\_out, z\_out

def plot\_plasma(self):

P.tricontourf(self.rzt[:, 0], self.rzt[:, 1],

self.tris, self.beta, 1001, zorder=0)

cticks = P.linspace(0.0, 0.2, 5)

P.colorbar(ticks=cticks, format='%.2f')

P.jet()

## Appendix G

# -\*- coding: utf-8 -\*-

"""

Created on Sat Apr 12 18:37:00 2014

@author: zchmlk

"""

from scipy.constants import pi

import pylab as P

import gspread

'''

Constructs coils and their corresponding flux loops

Writes each coil to the specified google spreadsheet with each

row representing a coil and each column representing a coil parameter

\*\*\*gspread must be installed on your device in order to run this script\*\*\*

'''

# dtype definitions for arrays (change here will affect all)

fdtype = P.float64

idtype = P.int16

# minor radius

a = 1.0

# major radius

r\_0 = 1.5

# radius of vacuum vessel (from r\_0), plus blanket if present

blanket\_width = 0.0

r\_vv0 = a + blanket\_width

# radius of flux conserver to Uberplate feature

a\_u = 0.242

# width of Uberplate

w\_u = 0.35

# width of injector

w\_i = 0.28

# r location of Uberplate (injector feature is perpendicular

# to Uberplate and tangent to the FC)

r\_u = r\_0 + P.sqrt((2 \* a - w\_u) \* a\_u - w\_u \*\* 2 / 4 + a \*\* 2)

# angle of mating point of FC and Uberplate feature

theta\_u = P.arctan2(0.5 \* w\_u + a\_u, a + a\_u)

# center of upper FC to Uberplate feature arc

r\_u\_1 = r\_u

z\_u\_1 = w\_u / 2 + a\_u

# center of lower FC to Uberplate feature arc

r\_u\_2 = r\_u

z\_u\_2 = -(w\_u / 2 + a\_u)

# coil parameters

coil\_width = 0.1

coils\_list = []

r\_coils\_list = []

z\_coils\_list = []

r\_floop\_list = []

z\_floop\_list = []

array\_radius = 1.15

array\_r\_width = 0.1

array\_z\_width = 0.1

delta\_theta\_degrees = 15.0

delta\_theta = P.deg2rad(delta\_theta\_degrees)

n\_coils\_each\_side = 5

theta\_start\_degrees = 270.0

theta\_start = P.deg2rad(theta\_start\_degrees)

theta\_end = theta\_start + (n\_coils\_each\_side - 1) \* delta\_theta

for theta in P.linspace(theta\_start, theta\_end, n\_coils\_each\_side):

r\_coil = r\_0 + array\_radius \* P.cos(theta)

z\_coil = array\_radius \* P.sin(theta)

r\_coils\_list.append(r\_coil)

z\_coils\_list.append(z\_coil)

r\_floop = r\_0 + a \* P.cos(theta)

z\_floop = a \* P.sin(theta)

r\_floop\_list.append(r\_floop)

z\_floop\_list.append(z\_floop)

theta\_start\_degrees = 90.0 - (n\_coils\_each\_side - 1) \* delta\_theta\_degrees

theta\_start = P.deg2rad(theta\_start\_degrees)

theta\_end = P.deg2rad(90.0)

for theta in P.linspace(theta\_start, theta\_end, n\_coils\_each\_side):

r\_coil = r\_0 + array\_radius \* P.cos(theta)

z\_coil = array\_radius \* P.sin(theta)

r\_coils\_list.append(r\_coil)

z\_coils\_list.append(z\_coil)

r\_floop = r\_0 + a \* P.cos(theta)

z\_floop = a \* P.sin(theta)

r\_floop\_list.append(r\_floop)

z\_floop\_list.append(z\_floop)

# coils on cryoports

r\_coils\_list.extend([1.16 + 0.5 \* coil\_width, 1.16 + 0.5 \* coil\_width,

0.55 + coil\_width, 0.55 + coil\_width])

z\_coils\_list.extend([-1.1, 1.1, -1.1, 1.1])

# keep index for double-wide coils

i\_wide\_coil = len(z\_coils\_list) - 2

# Change coils on injector here

r\_inj\_coils = [2.9, 3.2, 3.5] \* 2

n\_inj\_coils = len(r\_inj\_coils)

z\_inj\_coil = 0.25

z\_inj\_coils = P.ones(n\_inj\_coils, dtype=fdtype) \* z\_inj\_coil

for ii in xrange(int(n\_inj\_coils / 2)):

z\_inj\_coils[ii] \*= -1.0

r\_coils\_list.extend(r\_inj\_coils)

z\_coils\_list.extend(z\_inj\_coils)

# coil at end of injector

inj\_end\_coil = False

if inj\_end\_coil:

r\_coils\_list.append(1.5 + 1.17 + 0.67 + coil\_width)

z\_coils\_list.append(0.0)

# list entry of arc coil positions

r\_a\_width = 0.2

z\_a\_width = r\_a\_width

# Change arc coil params here

r\_mod = 0.05 \* 0

z\_mod = 0.05 \* 0

r\_a\_coils = P.array([r\_u - r\_mod, r\_u - r\_mod], dtype=fdtype)

z\_a\_coils = P.array([0.5 \* w\_u + a\_u - z\_mod, -(0.5 \* w\_u + a\_u - z\_mod)],

dtype=fdtype)

# total number of arc coils

n\_a\_coils = len(r\_a\_coils)

# add arc coils to r\_coils, z\_coils

for ii in xrange(0, n\_a\_coils):

r\_coils\_list.append(r\_a\_coils[ii])

z\_coils\_list.append(z\_a\_coils[ii])

# make arrays out of lists

r\_coils = P.array(r\_coils\_list, dtype=fdtype)

z\_upper = r\_vv0 + 0.5 \* coil\_width

z\_inj\_delta = 0.1

z\_coils = P.array(z\_coils\_list, dtype=fdtype)

# total number of coils

n\_coils = len(r\_coils)

r\_widths = P.ones(n\_coils, dtype=fdtype) \* coil\_width

z\_widths = 1.0 \* r\_widths

n\_r\_filaments = P.ones(n\_coils, dtype=idtype) \* 10

n\_z\_filaments = 1 \* n\_r\_filaments

# double-wide inner coils

r\_widths[i\_wide\_coil:i\_wide\_coil + 2] \*= 2.0

n\_r\_filaments[i\_wide\_coil:i\_wide\_coil + 2] \*= 2

# flux loops

# common factor for coil flux loops

fl\_fact = 1.0 / P.sqrt((r\_coils - r\_0) \*\* 2 + z\_coils \*\* 2)

# one floop for each coil

r\_floops = P.zeros(n\_coils, dtype=fdtype)

z\_floops = 0.0 \* r\_floops

# coil flux loops

# all on FC on line from (r\_0, 0) to center of coil

r\_floops[0:n\_coils] = r\_0 + a \* (r\_coils - r\_0) \* fl\_fact

z\_floops[0:n\_coils] = a \* z\_coils \* fl\_fact

l\_inj = 0.67 - 0.5 \* w\_i

# put last flux loop at end of injector

if inj\_end\_coil:

for ii in xrange(n\_coils - n\_inj\_coils - 1, n\_coils - 1):

r\_floops[ii] = r\_coils[ii]

z\_floops[ii] = z\_coils[ii] / abs(z\_coils[ii]) \* 0.5 \* w\_i

r\_floops[n\_coils - 1] = r\_u + l\_inj + 0.5 \* w\_i

else:

for ii in xrange(n\_coils - n\_inj\_coils, n\_coils):

r\_floops[ii] = r\_coils[ii]

z\_floops[ii] = z\_coils[ii] / abs(z\_coils[ii]) \* 0.5 \* w\_i

# arc coil flux floops

# first two are 45 degrees into the arc transition to injectors

r\_floops[n\_coils - 2] = r\_u\_1 + a\_u \* P.cos(1.25 \* pi)

z\_floops[n\_coils - 2] = z\_u\_1 + a\_u \* P.sin(1.25 \* pi)

r\_floops[n\_coils - 1] = r\_u\_2 + a\_u \* P.cos(0.75 \* pi)

z\_floops[n\_coils - 1] = z\_u\_2 + a\_u \* P.sin(0.75 \* pi)

n\_floops = len(r\_floops)

# writes coil params to google spreadsheet

#@author: NJ

gc = gspread.login('hitsiexperiment@gmail.com', '2vinjcity')

spread\_sheet = gc.open("coil\_set\_spreadsheet")

worksheet = spread\_sheet.sheet1

for ii in xrange(2, worksheet.row\_count + 1):

worksheet.update\_cell(ii,2, r\_coils[ii - 2]);

worksheet.update\_cell(ii,3, z\_coils[ii - 2]);

worksheet.update\_cell(ii,4, r\_widths[ii - 2]);

worksheet.update\_cell(ii,5, z\_widths[ii - 2]);

worksheet.update\_cell(ii,6, n\_r\_filaments[ii - 2]);

worksheet.update\_cell(ii,7, n\_z\_filaments[ii - 2]);

worksheet.update\_cell(ii,8, r\_floops[ii - 2]);

worksheet.update\_cell(ii,9, z\_floops[ii - 2]);

    '''

## Appendix H

# -\*- coding: utf-8 -\*-

"""

Created on Sat May 17 23:47:54 2014

@author: zchmlk

"""

from coil\_sim\_object import CoilSim

print "Defining coilsim"

cs = CoilSim()

print "Reading spreadsheet for coil set"

# read the "standard" (RKT) Google spreadsheet coil set 'coil\_set\_spreadsheet'

cs.coil\_set.read\_cvs()

# this Google spreadsheet modifies the inner-most radius coils

#cs.coil\_set.read\_cvs(filename='coil\_set\_spreadsheet\_2')

print "Calculating psi"

cs.calc\_psi()

print "Plotting"

cs.plot()

## Appendix I

# -\*- coding: utf-8 -\*-

"""

Created on Sat Oct 04 14:05:40 2014

@author: zchmlk

"""

from coil\_object import Coil

from coil\_set\_object import CoilSet

from Tkinter import \*

import pylab as P

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

from matplotlib.figure import Figure

import sys

if sys.version\_info[0] < 3:

import Tkinter as Tk

else:

import tkinter as Tk

###############################################################################

fdtype = P.float64

idtype = P.int16

widthFactor = 300.0

heightFactor = 400.0

rectScaleFactor = 10.0 # 1 / rsf

idCoords = {}

canvasMade = False

contourf = False

def makeCanvas():

print "Creating mesh..."

global canvasMade

canvasMade = True

createCanvas.config(state='disabled')

global root

root = Tk.Tk()

root.wm\_title("Coil Code GUI")

root.resizable(0,0)

global width

width = (r\_range.get() \* widthFactor)

global height

height = (z\_range.get() \* heightFactor)

global cs

cs = CoilSet(coils=[])

global window

window = Figure(figsize=(width / 100.0, height / 100.0), dpi=100)

global canvas

canvas = FigureCanvasTkAgg(window, master=root)

window.canvas.get\_tk\_widget().bind("<Button-3>", addCoilEvent)

window.canvas.get\_tk\_widget().bind("<Button-1>", removeCoilEvent)

makeSubPlot()

canvas.show()

canvas.get\_tk\_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

canvas.\_tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

root.protocol("WM\_DELETE\_WINDOW", on\_quit)

print "Right click to add coil with selected parameters"

print "Left click to remove coil(s)"

print "Flux contours will be plotted dynamically"

print "Exit mesh window in order to create new mesh"

def makeSubPlot():

global subPlot

subPlot = window.add\_subplot(111)

subPlot.set\_title('Coil Code GUI')

subPlot.set\_xlabel('R Coil')

subPlot.set\_xlim([0, r\_range.get()])

subPlot.set\_ylabel('Z Coil')

subPlot.set\_ylim([-z\_range.get(), z\_range.get()])

def addCoilEvent(event):

inv = subPlot.transData.inverted()

coordinates = inv.transform((event.x,event.y))

rPlot = coordinates[0]

zPlot = -coordinates[1]

if rPlot < r\_range.get() and rPlot > 0.0 and zPlot > -z\_range.get() and zPlot < z\_range.get():

addCoil(event.x, event.y)

else:

pass

def addCoil(r = 0.0, z = 0.0):

inv = subPlot.transData.inverted()

coordinates = inv.transform((r, z))

rPlot = coordinates[0]

zPlot = -coordinates[1]

coil = Coil(r\_coil = rPlot, z\_coil = zPlot, i\_coil =

i\_coil.get(), r\_width = r\_width.get(), z\_width = z\_width.get(),

n\_r\_filaments = n\_r\_filaments.get(), n\_z\_filaments = n\_z\_filaments.get())

cs.add\_Coil(coil)

plotFieldContours()

coordinates = subPlot.transData.transform((rPlot, zPlot))

coilNum = canvas.get\_tk\_widget().create\_rectangle(coordinates[0] - (width / rectScaleFactor) \* r\_width.get() / 2, z -

(height / rectScaleFactor) \* z\_width.get() / 2 , coordinates[0] +

(width / rectScaleFactor) \* r\_width.get() / 2, z + (height / rectScaleFactor)

\* z\_width.get() / 2, fill="red")

global coilCoords

coilCoords = str(rPlot) + "," + str(zPlot) + "," + str(r\_width.get()) + "," + str(z\_width.get())

global idCoords

idCoords[coilNum] = coilCoords

print "Coil added"

def removeCoilEvent(event):

removeCoil(event.x, event.y)

def removeCoil(r = 0.0, z = 0.0):

coilsToRemove = canvas.get\_tk\_widget().find\_overlapping(r - r\_width.get()

/ 200.0, z -

z\_width.get() / 200.0 , r +

r\_width.get() / 200.0, z +

z\_width.get() / 200.0)

for coil in coilsToRemove:

if idCoords.has\_key(coil):

canvas.get\_tk\_widget().delete(coil)

coords = idCoords[coil].split(",")

idCoords[coil] = None

r = coords[0]

z = coords[1]

for setcoil in cs.coils:

coilR = str(setcoil.r\_coil)

coilZ = str(setcoil.z\_coil)

if r == coilR and z == coilZ:

print "Coil removed"

cs.remove\_Coil(setcoil)

setcoil = None;

else:

pass

plotFieldContours()

def plotFieldContours():

window.clear()

makeSubPlot()

cs.calc\_fields(r\_range=P.array([1e-6, r\_range.get()],

dtype=fdtype),

z\_range=P.array([-z\_range.get(), z\_range.get()],

dtype=fdtype), r\_dim=r\_dims.get(), z\_dim=z\_dims.get())

if contourf\_var.get() and cs.n\_coils != 0:

cp = subPlot.contourf(cs.r\_arr, cs.z\_arr, cs.psi\_cumulative)

makeColorBar(cp)

elif cs.n\_coils != 0:

cp = subPlot.contour(cs.r\_arr, cs.z\_arr, cs.psi\_cumulative)

makeColorBar(cp)

canvas.show()

def makeColorBar(cp = None):

cb = window.colorbar(cp)

cb.set\_label("Psi Levels")

def on\_quit():

createCanvas.config(state='normal')

global canvasMade

canvasMade = False

root.destroy()

def on\_quit\_parameters():

if "cs" in globals():

print "Access coil set attributes using cs. notation"

print "Number of coils: cs.n\_coils = " + str(cs.n\_coils)

if canvasMade:

root.destroy()

master.destroy()

###############################################################################

'''Code for mesh/coil parameters window'''

print "Input mesh and coil parameters"

master = Tk.Tk()

master.wm\_title("Parameters")

hit\_pop\_var = IntVar()

contourf\_var = IntVar()

scrollBarSize = 250.0

def hit\_pop():

if hit\_pop\_var.get():

r\_range.set(4.0)

z\_range.set(1.5)

r\_dims.set(100)

z\_dims.set(100)

i\_coil.set(1.0)

r\_width.set(0.1)

z\_width.set(0.1)

n\_r\_filaments.set(10)

n\_z\_filaments.set(10)

hit\_pop\_button = Checkbutton(master, text="Hit-Pop Settings?", command=hit\_pop, variable=hit\_pop\_var)

hit\_pop\_button.pack()

contourf\_button = Checkbutton(master, text="Filled Contours?", variable=contourf\_var)

contourf\_button.pack()

r\_range = Scale(master, length= scrollBarSize, label="r\_range (0 to r\_range)", from\_=0, to=10,

resolution=0.05, orient=HORIZONTAL, fg="black", relief=RAISED,

highlightbackground="black",

font="helvetica", activebackground="yellow")

r\_range.pack()

z\_range = Scale(master, length= scrollBarSize, label="z\_range (-z\_range to z\_range)", from\_=0, to=5,

resolution=0.05, orient=HORIZONTAL, fg="black", relief=RAISED,

highlightbackground="black", activebackground="yellow",

font="helvetica")

z\_range.pack()

r\_dims = Scale(master, length= scrollBarSize, label="r\_dim", from\_=0, to=300,

resolution=1, orient=HORIZONTAL, fg="black", relief=RAISED,

highlightbackground="black",

font="helvetica", activebackground="yellow")

r\_dims.pack()

z\_dims = Scale(master, length= scrollBarSize, label="z\_dim", from\_=0, to=300,

resolution=1, orient=HORIZONTAL, fg="black", relief=RAISED,

highlightbackground="black", activebackground="yellow",

font="helvetica")

z\_dims.pack()

createCanvas = Button(master, text="Make Mesh", command=makeCanvas, fg="black",

highlightbackground="black", relief=RAISED,

font="helvetica", activebackground="yellow")

createCanvas.pack()

i\_coil = Scale(master, length= scrollBarSize, label="current", from\_=0, to=10,

resolution=0.05, orient=HORIZONTAL, fg="black", relief=RAISED,

highlightbackground="black",

font="helvetica", activebackground="yellow")

i\_coil.pack()

r\_width = Scale(master, length= scrollBarSize, label="r\_width", from\_=0, to=1,

resolution=0.05, orient=HORIZONTAL, fg="black", relief=RAISED,

highlightbackground="black", activebackground="yellow",

font="helvetica")

r\_width.pack()

z\_width = Scale(master, length= scrollBarSize, label="z\_width", from\_=0, to=1,

resolution=0.05, orient=HORIZONTAL, fg="black", relief=RAISED,

highlightbackground="black",

font="helvetica", activebackground="yellow")

z\_width.pack()

n\_r\_filaments = Scale(master, length= scrollBarSize, label="n\_r\_filaments",

from\_=0, to=100, orient=HORIZONTAL, fg="black",

relief=RAISED, highlightbackground="black",

font="helvetica", activebackground="yellow")

n\_r\_filaments.pack()

n\_z\_filaments = Scale(master, length= scrollBarSize, label="n\_z\_filaments",

from\_=0, to=100, orient=HORIZONTAL, fg="black",

highlightbackground="black", relief=RAISED,

font="helvetica", activebackground="yellow")

n\_z\_filaments.pack()

master.protocol("WM\_DELETE\_WINDOW", on\_quit\_parameters)

Tk.mainloop()