18.335Fin lProject_1.1

M y 16, 2023

0.1 18.335 Fin l Project - Comp rison of 2D Diffusion with MC using IPI nd LOBPCG Iter tive solvers for k-eigenv lue c lcul tions.

0.1.1 Section I: Gener te reference Monte C rlo Solution nd Gener te Cross Sections for use in the 2D Diffusion Solver

[1]:
```
#Created by: J. Sebastian Tchakerian
#Project For: 18.335
#Date due: 05/16/2023

%m tplotlib inline

import numpy  s np
from uncert inties import ufloat
from m tplotlib import pyplot  s plt
import p nd s  s pd
from scipy.sp rse.lin lg import lobpcg
from pyp pi import events, papi_high  s high
import openmc
import w rnings
warnings.simplefilter(action='ignore', category=FutureW rning) #Ignore annoying␣
  future warning message
import p ndoc
```

[2]:
```
uo2 = openmc.Material(name='fuel')
uo2.add_element('U', 1, enrichment=3.2)
uo2.add_element('O', 2)
uo2.set_density('g/cc', 10.341)
```

[3]:
```
water = openmc.model.borated_water(400)
```

[4]:
```
materials = openmc.Materials([uo2, water])
```

[5]:
```
materials.export_to_xml()
```

[6]:
```
#fuel pin dimensions, 0.4 cm sqaure of fuel in 1.2 cm square moderator
side_fuel = 0.4
pitch = 1.2
```

[7]:
```
# square fuel in square box, vacuum boundaries were use on the outside for␣
  calculating CPs
rfo = openmc.model.rectangular_prism(side_fuel,side_fuel)
xy_box = openmc.model.rectangular_prism(3*pitch, 3*pitch,
        boundary_type='reflective')
z0 = openmc.ZPlane(z0=-10, boundary_type='reflective')
z1 = openmc.ZPlane(z0=10, boundary_type='reflective')
```

[8]:
```
rfo
```

[8]:
```
<openmc.region.Intersection at 0x7fb93631ff90>
```

[9]:
```
fuel = openmc.Cell(cell_id=1, name='fuel', fill=uo2)
fuel.region = rfo
mod = openmc.Cell(cell_id=2, name='moderator', fill=water)
mod.region = ~rfo
fuel_univ = openmc.Universe(cells=(fuel, mod))
```

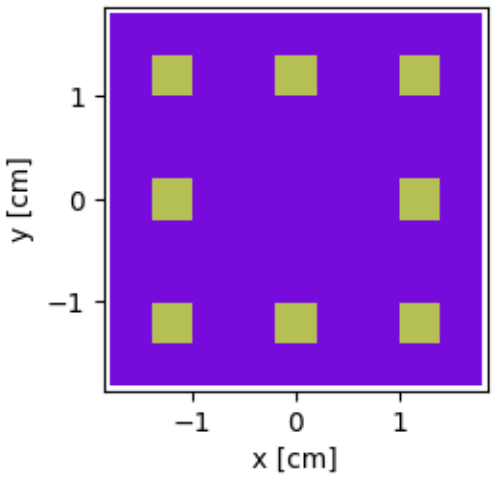
[10]:
```
mod = openmc.Cell(cell_id=3, name='guide', fill=water)
guide_univ = openmc.Universe(cells=(mod, ))
```

[11]:
```
# Build lattice with 8 fuel pins and 1 center empty guide tube
lat = openmc.RectLattice()
lat.pitch = [pitch, pitch]
lat.lower_left = [-1.5*pitch, -1.5*pitch]
lat.universes = [[fuel_univ, fuel_univ, fuel_univ],
                 [fuel_univ, guide_univ, fuel_univ],
                 [fuel_univ, fuel_univ, fuel_univ]]
```

[12]:
```
root_cell = openmc.Cell(fill=lat)
root_cell.region = xy_box & +z0 & -z1
root = openmc.Universe(cells=(root_cell, ))
geometry = openmc.Geometry(root)
geometry.export_to_xml()
```

[13]:
```
root.plot(width=(3.1*pitch, 3.1*pitch), color_by='material')
```

[13]:
```
<matplotlib.image. xesImage at 0x7fb9363d7ed0>
```



[14]:
```
lat
```

[14]:
```
RectLattice
        ID             =       3
        Name           =
        Shape          =       (3, 3)
        Lower Left     =       [-1.7999999999999998, -1.7999999999999998]
        Pitch          =       [1.2, 1.2]
        Outer          =       None
        Universes
1 1 1
1 2 1
1 1 1
```

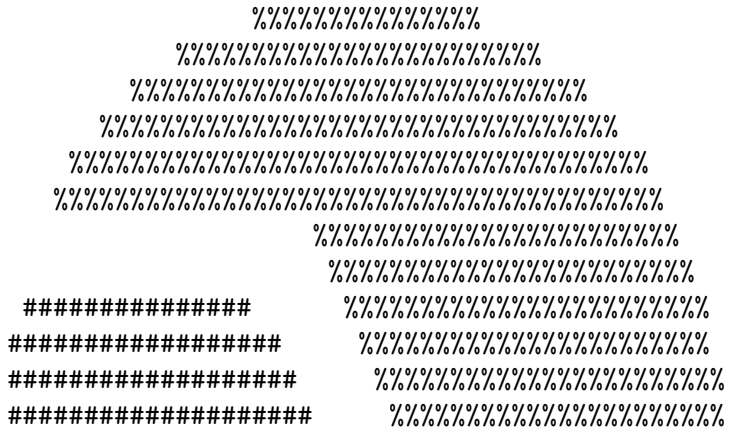
[15]:
```
groups = openmc.mgxs.EnergyGroups((0.0, 20.0e6)) #MONOENERGETIC - NECESS RY FOR␣
   LOBPCG

# Instantiate an MGXS library.
mgxs_lib = openmc.mgxs.Library(geometry)
mgxs_lib.energy_groups = groups

# Don't apply any anisotropic scattering corrections.
mgxs_lib.correction = None

# Set the desired MGXS data.
mgxs_lib.mgxs_types = ('total', 'absorption', 'nu-fission', 'scatter matrix',␣
   'chi')

# Define the domain and build the library.
mgxs_lib.domain_type = 'cell'
```

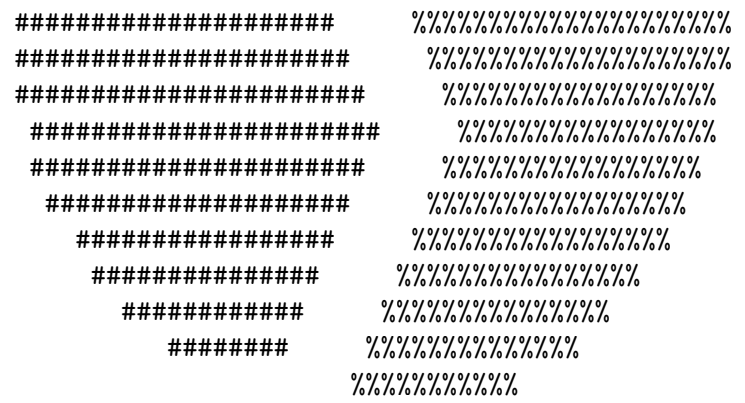
```
mgxs_lib.domains = geometry.get_all_material_cells().values()
mgxs_lib.build_library()

#  dd the tallies.
tallies = openmc.Tallies()
mgxs_lib.add_to_tallies_file(tallies)

tallies.export_to_xml()
#Warning is not an issue.
```

```
/home/sebastian/.local/lib/python3.11/site-packages/openmc/mixin.py:70:
IDWarning:  nother Filter instance already exists with id=48.
  warn(msg, IDWarning)
/home/sebastian/.local/lib/python3.11/site-packages/openmc/mixin.py:70:
IDWarning:  nother Filter instance already exists with id=2.
  warn(msg, IDWarning)
/home/sebastian/.local/lib/python3.11/site-packages/openmc/mixin.py:70:
IDWarning:  nother Filter instance already exists with id=9.
  warn(msg, IDWarning)
```

[16]:
```
settings = openmc.Settings()

settings.batches = 100
settings.inactive = 20
settings.particles = 10000

# only_fissionable performs a rejection sampling and only accepts source␣
  neutrons in material with fission xs
space = openmc.stats.Box((-1.5*pitch, -1.5*pitch, 0),
     (1.5*pitch, 1.5*pitch, 0), only_fissionable=True)
settings.source = openmc.Source(space=space)

settings.export_to_xml()
```

[17]:
```
openmc.run()
```





```
              | The OpenMC Monte Carlo Code
    Copyright | 2011-2023 MIT, UChicago  rgonne LLC, and contributors
      License | https://docs.openmc.org/en/latest/license.html
      Version | 0.13.4-dev
     Git SH 1 | 65618384c926d5c047460dac2c83db4a2de17915
    Date/Time | 2023-05-16 12:21:51
OpenMP Threads | 16

Reading settings XML file…
Reading cross sections XML file…
Reading materials XML file…
Reading geometry XML file…
Reading U234 from /home/sebastian/Codes/NuclearData/mcnp_endfb70/U234.h5
Reading U235 from /home/sebastian/Codes/NuclearData/mcnp_endfb70/U235.h5
Reading U238 from /home/sebastian/Codes/NuclearData/mcnp_endfb70/U238.h5
Reading U236 from /home/sebastian/Codes/NuclearData/mcnp_endfb70/U236.h5
Reading O16 from /home/sebastian/Codes/NuclearData/mcnp_endfb70/O16.h5
Reading O17 from /home/sebastian/Codes/NuclearData/mcnp_endfb70/O17.h5
Reading H1 from /home/sebastian/Codes/NuclearData/mcnp_endfb70/H1.h5
Reading H2 from /home/sebastian/Codes/NuclearData/mcnp_endfb70/H2.h5
Reading B10 from /home/sebastian/Codes/NuclearData/mcnp_endfb70/B10.h5
Reading B11 from /home/sebastian/Codes/NuclearData/mcnp_endfb70/B11.h5
Reading c_H_in_H2O from
/home/sebastian/Codes/NuclearData/mcnp_endfb70/c_H_in_H2O.h5
Minimum neutron data temperature: 294 K
Maximum neutron data temperature: 294 K
Reading tallies XML file…
Preparing distributed cell instances…
Reading plot XML file…
Writing summary.h5 file…
Maximum neutron transport energy: 20000000 eV for U235
Initializing source particles…

====================>     K EIGENV LUE SIMUL TION     <====================

 Bat./Gen.      k             verage k
```

1/1	1.00462	
2/1	1.01678	
3/1	1.03124	
4/1	0.99776	
5/1	1.00481	
6/1	1.00457	
7/1	1.00443	
8/1	0.98781	
9/1	1.00307	
10/1	0.99636	
11/1	1.03871	
12/1	1.00647	
13/1	1.00748	
14/1	0.99221	
15/1	1.00712	
16/1	1.01241	
17/1	1.02323	
18/1	1.00983	
19/1	1.02200	
20/1	1.00076	
21/1	1.00393	
22/1	0.99951	1.00172 +/- 0.00221
23/1	1.01360	1.00568 +/- 0.00416
24/1	1.01431	1.00784 +/- 0.00365
25/1	1.00660	1.00759 +/- 0.00284
26/1	1.03317	1.01185 +/- 0.00485
27/1	1.01290	1.01200 +/- 0.00410
28/1	1.00045	1.01056 +/- 0.00384
29/1	1.00325	1.00975 +/- 0.00348
30/1	1.01673	1.01044 +/- 0.00319
31/1	1.02695	1.01194 +/- 0.00325
32/1	1.00413	1.01129 +/- 0.00304
33/1	1.01080	1.01126 +/- 0.00280
34/1	0.98771	1.00957 +/- 0.00309
35/1	1.01533	1.00996 +/- 0.00290
36/1	0.99748	1.00918 +/- 0.00282
37/1	0.99482	1.00833 +/- 0.00278
38/1	0.99530	1.00761 +/- 0.00272
39/1	1.01793	1.00815 +/- 0.00263
40/1	1.01553	1.00852 +/- 0.00252
41/1	0.99276	1.00777 +/- 0.00251
42/1	1.01200	1.00796 +/- 0.00240
43/1	0.99685	1.00748 +/- 0.00235
44/1	1.02391	1.00816 +/- 0.00235
45/1	0.98930	1.00741 +/- 0.00238
46/1	0.99557	1.00695 +/- 0.00233
47/1	0.99822	1.00663 +/- 0.00226

```
48/1    1.03247    1.00755 +/- 0.00237
49/1    1.00751    1.00755 +/- 0.00229
50/1    1.00108    1.00734 +/- 0.00222
51/1    1.02102    1.00778 +/- 0.00219
52/1    0.99754    1.00746 +/- 0.00215
53/1    1.01123    1.00757 +/- 0.00208
54/1    1.00875    1.00761 +/- 0.00202
55/1    1.02612    1.00814 +/- 0.00203
56/1    0.99269    1.00771 +/- 0.00202
57/1    1.01050    1.00778 +/- 0.00197
58/1    1.01187    1.00789 +/- 0.00192
59/1    1.01449    1.00806 +/- 0.00188
60/1    0.99945    1.00784 +/- 0.00184
61/1    1.00259    1.00772 +/- 0.00180
62/1    1.00627    1.00768 +/- 0.00176
63/1    0.99675    1.00743 +/- 0.00173
64/1    0.98964    1.00702 +/- 0.00174
65/1    0.99762    1.00681 +/- 0.00172
66/1    1.00455    1.00676 +/- 0.00168
67/1    1.00294    1.00668 +/- 0.00164
68/1    1.01911    1.00694 +/- 0.00163
69/1    1.00869    1.00698 +/- 0.00160
70/1    1.00831    1.00700 +/- 0.00156
71/1    1.01489    1.00716 +/- 0.00154
72/1    0.99761    1.00698 +/- 0.00152
73/1    1.00360    1.00691 +/- 0.00150
74/1    0.98926    1.00658 +/- 0.00150
75/1    1.00742    1.00660 +/- 0.00148
76/1    1.00908    1.00664 +/- 0.00145
77/1    1.00864    1.00668 +/- 0.00142
78/1    0.99984    1.00656 +/- 0.00140
79/1    1.01912    1.00677 +/- 0.00140
80/1    1.00405    1.00673 +/- 0.00137
81/1    1.01118    1.00680 +/- 0.00135
82/1    1.00687    1.00680 +/- 0.00133
83/1    1.00206    1.00673 +/- 0.00131
84/1    0.98055    1.00632 +/- 0.00136
85/1    0.99456    1.00614 +/- 0.00135
86/1    1.02066    1.00636 +/- 0.00134
87/1    1.01364    1.00647 +/- 0.00133
88/1    1.00476    1.00644 +/- 0.00131
89/1    0.99642    1.00630 +/- 0.00130
90/1    1.00074    1.00622 +/- 0.00128
91/1    1.00862    1.00625 +/- 0.00126
92/1    1.00896    1.00629 +/- 0.00125
93/1    0.99345    1.00611 +/- 0.00124
94/1    1.00100    1.00604 +/- 0.00123
95/1    1.02401    1.00628 +/- 0.00123
```

```
      96/1    0.99541    1.00614 +/- 0.00123
      97/1    0.99790    1.00603 +/- 0.00121
      98/1    1.01360    1.00613 +/- 0.00120
      99/1    1.01642    1.00626 +/- 0.00119
     100/1    0.99037    1.00606 +/- 0.00120
Creating state point statepoint.100.h5…

=======================>     TIMING ST TISTICS     <=======================

Total time for initialization     = 2.2616e-01 seconds
  Reading cross sections          = 2.1568e-01 seconds
Total time in simulation          = 1.4613e+01 seconds
  Time in transport only          = 1.4322e+01 seconds
  Time in inactive batches        = 1.7394e+00 seconds
  Time in active batches          = 1.2874e+01 seconds
  Time synchronizing fission bank = 5.5763e-02 seconds
    Sampling source sites         = 5.2151e-02 seconds
    SEND/RECV source sites        = 3.5634e-03 seconds
  Time accumulating tallies       = 2.0290e-01 seconds
  Time writing statepoints        = 3.3325e-03 seconds
Total time for finalization       = 9.5557e-05 seconds
Total time elapsed                = 1.4854e+01 seconds
Calculation Rate (inactive)       = 114983 particles/second
Calculation Rate (active)         = 62141.8 particles/second

============================>     RESULTS     <============================

k-effective (Collision)     = 1.00681 +/- 0.00098
k-effective (Track-length)  = 1.00606 +/- 0.00120
k-effective ( bsorption)    = 1.00787 +/- 0.00113
Combined k-effective        = 1.00704 +/- 0.00081
Leakage Fraction            = 0.00000 +/- 0.00000
```

[18]:
```
# Load the statepoint and the MGXS results.
sp = openmc.StatePoint('statepoint.100.h5')
mgxs_lib.load_from_statepoint(sp)
```

[19]:
```
# Pick out the fuel and moderator domains.
fuel = mgxs_lib.domains[0]
moderator = mgxs_lib.domains[1]
center = mgxs_lib.domains[2]
 ssert fuel.name == 'fuel'
 ssert moderator.name == 'moderator'
 ssert center.name == 'guide'
```

[20]:
```
#Obtain data in array form for necessary components using dataframes

df = mgxs_lib.get_mgxs(fuel, 'total').get_pandas_dataframe()
print(df)
FuelXS_Total = df["mean"]
Fuel_D = (1/(3*FuelXS_Total))
print(Fuel_D)

df = mgxs_lib.get_mgxs(moderator, 'total').get_pandas_dataframe()
print(df)
ModeratorXS_Total = df["mean"]
Moderator_D = (1/(3*ModeratorXS_Total))
print(Moderator_D)

df = mgxs_lib.get_mgxs(center, 'total').get_pandas_dataframe()
print(df)
CenterXS_Total = df["mean"]
Center_D = (1/(3*CenterXS_Total))

df = mgxs_lib.get_mgxs(fuel, 'absorption').get_pandas_dataframe()
print(df)
FuelXS_ bsorption = df["mean"]

df = mgxs_lib.get_mgxs(moderator, 'absorption').get_pandas_dataframe()
print(df)
ModeratorXS_ bsorption = df["mean"]

df = mgxs_lib.get_mgxs(center, 'absorption').get_pandas_dataframe()
print(df)
CenterXS_ bsorption = df["mean"]

df = mgxs_lib.get_mgxs(fuel, 'nu-fission').get_pandas_dataframe()
print(df)
FuelXS_NuF = df["mean"]

df = mgxs_lib.get_mgxs(fuel, 'chi').get_pandas_dataframe()
Fuel_Chi = df["mean"]
print(df)

df = mgxs_lib.get_mgxs(fuel, 'scatter matrix').get_pandas_dataframe()
print(df)
#FuelXS_Scatter = df["mean"]

df = mgxs_lib.get_mgxs(moderator, 'scatter matrix').get_pandas_dataframe()
ModeratorXS_Scatter = df["mean"]
print(df)
```

```
df = mgxs_lib.get_mgxs(center, 'scatter matrix').get_pandas_dataframe()
CenterXS_Scatter = df["mean"]
```

```
   cell  group in nuclide      mean  std. dev.
0     1         1   total  0.554752   0.000517
0    0.600869
Name: mean, dtype: float64
   cell  group in nuclide      mean  std. dev.
0     2         1   total  1.715762   0.001661
0    0.194277
Name: mean, dtype: float64
   cell  group in nuclide      mean  std. dev.
0     3         1   total  1.787939   0.002463
   cell  group in nuclide      mean  std. dev.
0     1         1   total  0.167347   0.000208
   cell  group in nuclide      mean  std. dev.
0     2         1   total  0.012501   0.000016
   cell  group in nuclide      mean  std. dev.
0     3         1   total  0.013417   0.000025
   cell  group in nuclide      mean  std. dev.
0     1         1   total  0.282886   0.000381
   cell  group out nuclide  mean  std. dev.
0     1          1   total   1.0   0.001647
   cell  group in  group out nuclide      mean  std. dev.
0     1         1          1   total  0.387552   0.000586
   cell  group in  group out nuclide     mean  std. dev.
0     2         1          1   total  1.70359   0.001653
```

[21]:
```
'''
Variables are called:

FuelXS_Total
Fuel_D
FuelXS_ bsorption
FuelXS_NuF
Fuel_Chi
ModeratorXS_Total
ModeratorXS_ bsorption
Moderator_D
CenterXS_Total
CenterXS_ bsorption
Center_D

'''

#Create function to autogenerate approporiate d-tilde
```

```
def dtilde(d1,d2,spacing):
    dtilde = 2 * d1 * d2/(spacing*(d1+d2))
    return dtilde

#---Generate problem mesh---

xWidth = 3.6
yWidth = 3.6
spacing = 0.4 #Must be equal to 0.4 or less by even amounts (0.2,0.1 etc)
x_rows = int(xWidth/spacing)
y_columns = int(yWidth/spacing)
num_cells = x_rows * y_columns
n_groups = 1 #1Group model

#Create a class for the cells

cl ss cells:
    def __init__(self, material):
        self.material = material
    def matID(self):
        return self.material

#Use cell class to fill in an array of cell information. This will create a␣
   list of strings which correspond to the material of each cell

Cell_Info = []

#Cursor for moving through the geometry

xpos = spacing/2
ypos = spacing/2

for i in range(num_cells):
    if 2.0>xpos>1.6  nd 2.0>ypos>1.6:
        Cell_Info.append(cells('Center'))
    elif (0.8>xpos>0.4 or 2.0>xpos>1.6 or 3.2>xpos>2.8)  nd (0.8>ypos>0.4 or 2.
  0>ypos>1.6 or 3.2>ypos>2.8):
        Cell_Info.append(cells('Fuel'))
    else:
        Cell_Info.append(cells('Moderator'))

    xpos = xpos + spacing #Move along x

    if xpos > 3.6: #If the x cursor needs to be reset, reset it
        xpos = spacing/2
        ypos = ypos + spacing #Move the y cursor for the next row of cells
```

```
#Instantiate Matricies

D_Matrix = np.zeros([n_groups*num_cells, n_groups*num_cells])
F_Matrix = np.zeros([n_groups*num_cells, n_groups*num_cells])


#---Fill in the matricies---

#Generate midpoint cursor used to track geometric position while filling in␣
  matrix

xpos = spacing/2
ypos = spacing/2

for i in range(num_cells):
    if xpos<spacing  nd ypos<spacing: #Corner cell 1
        DT_T = dtilde(Moderator_D, Moderator_D, spacing)
        DT_R = dtilde(Moderator_D, Moderator_D, spacing)
        DT_L = 0
        DT_B = 0
        D_Matrix[i,i] = ModeratorXS_ bsorption * (spacing**2) + DT_T + DT_B +␣
  DT_R + DT_L
        D_Matrix[i,i+1] = -DT_R
        D_Matrix[i,i+y_columns] = -DT_T
        F_Matrix[i,i] = 0
    elif xpos>(xWidth-spacing)  nd ypos<spacing: #Corner cell 2
        DT_T = dtilde(Moderator_D, Moderator_D, spacing)
        DT_R = 0
        DT_L = dtilde(Moderator_D, Moderator_D, spacing)
        DT_B = 0
        D_Matrix[i,i] = ModeratorXS_ bsorption * (spacing**2) + DT_T + DT_B +␣
  DT_R + DT_L
        D_Matrix[i,i-1] = -DT_L
        D_Matrix[i,i+y_columns] = -DT_T
        F_Matrix[i,i] = 0
    elif xpos<spacing  nd ypos>(yWidth-spacing): #Corner cell 3
        DT_T = 0
        DT_R = dtilde(Moderator_D, Moderator_D, spacing)
        DT_L = 0
        DT_B = dtilde(Moderator_D, Moderator_D, spacing)
        D_Matrix[i,i] = ModeratorXS_ bsorption * (spacing**2) + DT_T + DT_B +␣
  DT_R + DT_L
        D_Matrix[i,i+1] = -DT_R
        D_Matrix[i,i-y_columns] = -DT_B
        F_Matrix[i,i] = 0
    elif xpos>(xWidth-spacing)  nd ypos>(yWidth-spacing): #Corner cell 4
        DT_T = 0
```

```
      DT_R = 0
      DT_L = dtilde(Moderator_D, Moderator_D, spacing)
      DT_B = dtilde(Moderator_D, Moderator_D, spacing)
      D_Matrix[i,i] = ModeratorXS_ bsorption * (spacing**2) + DT_T + DT_B +␣
DT_R + DT_L
      D_Matrix[i,i-1] = -DT_L
      D_Matrix[i,i-y_columns] = -DT_B
      F_Matrix[i,i] = 0
  elif xpos>(spacing)  nd ypos<spacing: #Bottom Edges
      #Perform check on above cell to see if it is fuel
      Material = Cell_Info[i+y_columns].matID()
      if Material == 'Fuel':
          DT_T = dtilde(Moderator_D, Fuel_D, spacing)
      else:
          DT_T = dtilde(Moderator_D, Moderator_D,spacing)
      DT_R = dtilde(Moderator_D, Moderator_D, spacing)
      DT_L = dtilde(Moderator_D, Moderator_D, spacing)
      DT_B = 0
      D_Matrix[i,i] = ModeratorXS_ bsorption * (spacing**2) + DT_T + DT_B +␣
DT_R + DT_L
      D_Matrix[i,i+1] = -DT_R
      D_Matrix[i,i-1] = -DT_L
      D_Matrix[i,i+y_columns] = -DT_T
      F_Matrix[i,i] = 0
  elif xpos<(spacing)  nd ypos>spacing: #Left Edges
      #Perform check on right cell to see if it is fuel
      Material = Cell_Info[i+1].matID()
      if Material == 'Fuel':
          DT_R = dtilde(Moderator_D, Fuel_D, spacing)
      else:
          DT_R = dtilde(Moderator_D, Moderator_D,spacing)
      DT_T = dtilde(Moderator_D, Moderator_D, spacing)
      DT_L = 0
      DT_B = dtilde(Moderator_D,Moderator_D,spacing)
      D_Matrix[i,i] = ModeratorXS_ bsorption * (spacing**2) + DT_T + DT_B +␣
DT_R + DT_L
      D_Matrix[i,i+1] = -DT_R
      D_Matrix[i,i-y_columns] = -DT_B
      D_Matrix[i,i+y_columns] = -DT_T
      F_Matrix[i,i] = 0
  elif xpos>(xWidth-spacing)  nd ypos>spacing: #Right Edges
      #Perform check on left cell to see if it is fuel
      Material = Cell_Info[i-1].matID()
      if Material == 'Fuel':
          DT_L = dtilde(Moderator_D, Fuel_D, spacing)
      else:
          DT_L = dtilde(Moderator_D, Moderator_D,spacing)
```

```
      DT_T = dtilde(Moderator_D, Moderator_D, spacing)
      DT_R = 0
      DT_B = dtilde(Moderator_D,Moderator_D,spacing)
      D_Matrix[i,i] = ModeratorXS_ bsorption * (spacing**2) + DT_T + DT_B +␣
DT_R + DT_L
      D_Matrix[i,i-1] = -DT_L
      D_Matrix[i,i-y_columns] = -DT_B
      D_Matrix[i,i+y_columns] = -DT_T
      F_Matrix[i,i] = 0
  elif xpos>spacing  nd ypos>(yWidth-spacing): #Top Edges
      #Perform check on bottom cell to see if it is fuel
      Material = Cell_Info[i-y_columns].matID()
      if Material == 'Fuel':
          DT_B = dtilde(Moderator_D, Fuel_D, spacing)
      else:
          DT_B = dtilde(Moderator_D, Moderator_D,spacing)
      DT_T = 0
      DT_R = dtilde(Moderator_D, Moderator_D, spacing)
      DT_B = dtilde(Moderator_D,Moderator_D,spacing)
      DT_L = dtilde(Moderator_D,Moderator_D,spacing)
      D_Matrix[i,i] = ModeratorXS_ bsorption * (spacing**2) + DT_T + DT_B +␣
DT_R + DT_L
      D_Matrix[i,i-1] = -DT_L
      D_Matrix[i,i+1] = -DT_R
      D_Matrix[i,i-y_columns] = -DT_B
      F_Matrix[i,i] = 0
  else: #The cell is interior type
      Cell_Type = Cell_Info[i].matID()
      Cell_T = Cell_Info[i+y_columns].matID()
      Cell_B = Cell_Info[i-y_columns].matID()
      Cell_L = Cell_Info[i-1].matID()
      Cell_R = Cell_Info[i+1].matID()
      if(Cell_Type == 'Fuel'):
          if(Cell_T == 'Fuel'):
              DT_T = dtilde(Fuel_D, Fuel_D, spacing)
          elif(Cell_T == 'Moderator'):
              DT_T = dtilde(Fuel_D, Moderator_D, spacing)
          elif(Cell_T == 'Center'):
              DT_T = dtilde(Fuel_D, Center_D, spacing)
          if(Cell_B == 'Fuel'):
              DT_B = dtilde(Fuel_D, Fuel_D, spacing)
          elif(Cell_B == 'Moderator'):
              DT_B = dtilde(Fuel_D, Moderator_D, spacing)
          elif(Cell_B == 'Center'):
              DT_B = dtilde(Fuel_D, Center_D, spacing)
          if(Cell_L == 'Fuel'):
              DT_L = dtilde(Fuel_D, Fuel_D, spacing)
```

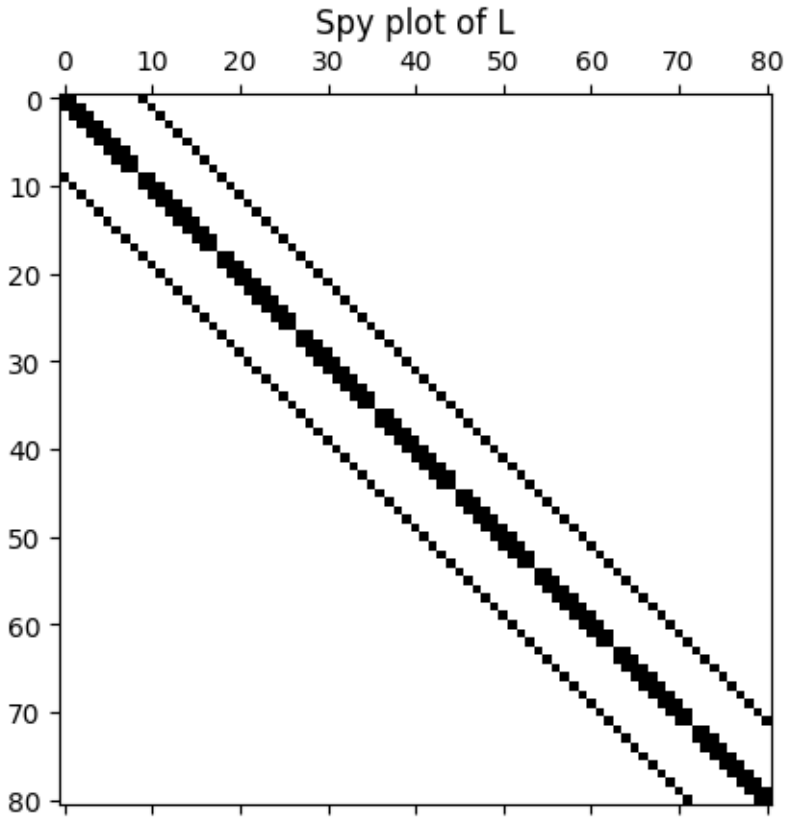
```
            elif(Cell_L == 'Moderator'):
                DT_L = dtilde(Fuel_D, Moderator_D, spacing)
            elif(Cell_L == 'Center'):
                DT_L = dtilde(Fuel_D, Center_D, spacing)
            if(Cell_R == 'Fuel'):
                DT_R = dtilde(Fuel_D, Fuel_D, spacing)
            elif(Cell_R == 'Moderator'):
                DT_R = dtilde(Fuel_D, Moderator_D, spacing)
            elif(Cell_R == 'Center'):
                DT_R = dtilde(Fuel_D, Center_D, spacing)
            D_Matrix[i,i] = FuelXS_ bsorption * (spacing**2) + DT_T + DT_B +␣
DT_R + DT_L
            D_Matrix[i,i-1] = -DT_L
            D_Matrix[i,i+1] = -DT_R
            D_Matrix[i,i-y_columns] = -DT_B
            D_Matrix[i,i+y_columns] = -DT_T
            F_Matrix[i,i] = Fuel_Chi * FuelXS_NuF * (spacing**2)
        if(Cell_Type == 'Moderator'):
            if(Cell_T == 'Fuel'):
                DT_T = dtilde(Moderator_D, Fuel_D, spacing)
            elif(Cell_T == 'Moderator'):
                DT_T = dtilde(Moderator_D, Moderator_D, spacing)
            elif(Cell_T == 'Center'):
                DT_T = dtilde(Moderator_D, Center_D, spacing)
            if(Cell_B == 'Fuel'):
                DT_B = dtilde(Moderator_D, Fuel_D, spacing)
            elif(Cell_B == 'Moderator'):
                DT_B = dtilde(Moderator_D, Moderator_D, spacing)
            elif(Cell_B == 'Center'):
                DT_B = dtilde(Moderator_D, Center_D, spacing)
            if(Cell_L == 'Fuel'):
                DT_L = dtilde(Moderator_D, Fuel_D, spacing)
            elif(Cell_L == 'Moderator'):
                DT_L = dtilde(Moderator_D, Moderator_D, spacing)
            elif(Cell_L == 'Center'):
                DT_L = dtilde(Moderator_D, Center_D, spacing)
            if(Cell_R == 'Fuel'):
                DT_R = dtilde(Moderator_D, Fuel_D, spacing)
            elif(Cell_R == 'Moderator'):
                DT_R = dtilde(Moderator_D, Moderator_D, spacing)
            elif(Cell_R == 'Center'):
                DT_R = dtilde(Moderator_D, Center_D, spacing)
            D_Matrix[i,i] = ModeratorXS_ bsorption * (spacing**2) + DT_T + DT_B␣
+ DT_R + DT_L
            D_Matrix[i,i-1] = -DT_L
            D_Matrix[i,i+1] = -DT_R
            D_Matrix[i,i-y_columns] = -DT_B
```

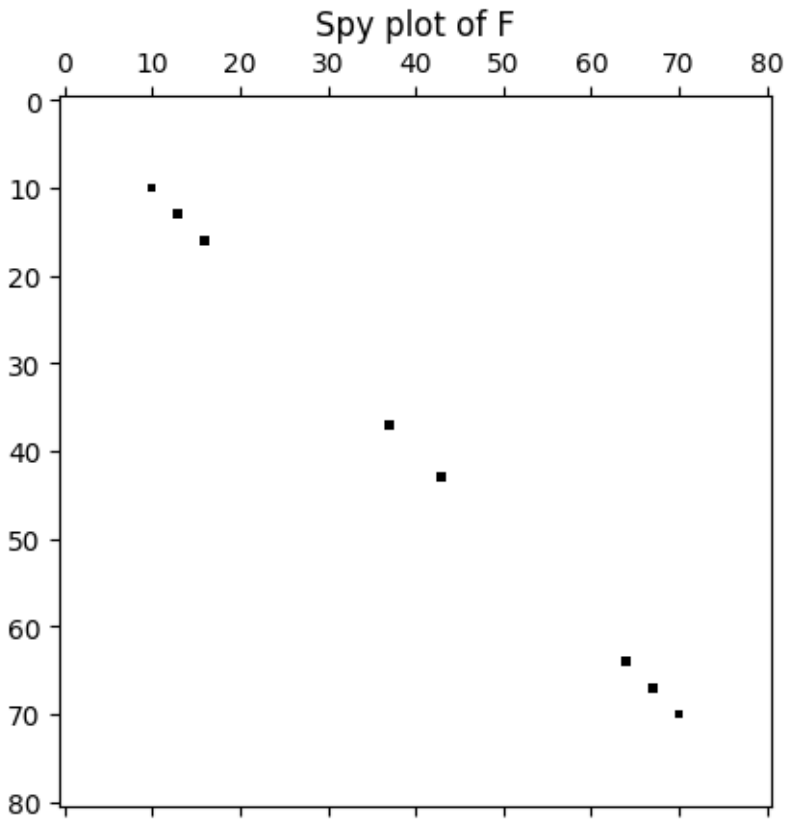
```
            D_Matrix[i,i+y_columns] = -DT_T
            F_Matrix[i,i] = 0
        if(Cell_Type == 'Center'):
            if(Cell_T == 'Fuel'):
                DT_T = dtilde(Center_D, Fuel_D, spacing)
            elif(Cell_T == 'Moderator'):
                DT_T = dtilde(Center_D, Moderator_D, spacing)
            elif(Cell_T == 'Center'):
                DT_T = dtilde(Center_D, Center_D, spacing)
            if(Cell_B == 'Fuel'):
                DT_B = dtilde(Center_D, Fuel_D, spacing)
            elif(Cell_B == 'Moderator'):
                DT_B = dtilde(Center_D, Moderator_D, spacing)
            elif(Cell_B == 'Center'):
                DT_B = dtilde(Center_D, Center_D, spacing)
            if(Cell_L == 'Fuel'):
                DT_L = dtilde(Center_D, Fuel_D, spacing)
            elif(Cell_L == 'Moderator'):
                DT_L = dtilde(Center_D, Moderator_D, spacing)
            elif(Cell_L == 'Center'):
                DT_L = dtilde(Center_D, Center_D, spacing)
            if(Cell_R == 'Fuel'):
                DT_R = dtilde(Center_D, Fuel_D, spacing)
            elif(Cell_R == 'Moderator'):
                DT_R = dtilde(Center_D, Moderator_D, spacing)
            elif(Cell_R == 'Center'):
                DT_R = dtilde(Center_D, Center_D, spacing)
            D_Matrix[i,i] = CenterXS_ bsorption * (spacing**2) + DT_T + DT_B +␣
  DT_R + DT_L
            D_Matrix[i,i-1] = -DT_L
            D_Matrix[i,i+1] = -DT_R
            D_Matrix[i,i-y_columns] = -DT_B
            D_Matrix[i,i+y_columns] = -DT_T
            F_Matrix[i,i] = 0

    #Update position of the x cursor

    xpos = xpos + spacing

    #Update position for new row if needed

    if (xpos > 3.6):
        xpos = spacing/2 #reset position of x cursor
        ypos = ypos + spacing #Move y cursor up one position

DebugFrame = pd.DataFrame(D_Matrix)
DebugFrame.to_csv('Matrix_Info.csv')
```

```
pd.DataFrame(F_Matrix).to_csv('1FissionMatrix.csv')
```

0.1.2 Section II: Gener te the 2D Diffusion Problem

[22]:
```
#---Construct Spy Plots of the Generated Matrices---

plt.figure(0)
plt.spy(D_Matrix)
plt.title('Spy plot of L')
plt.savefig('Spy_L.png', dpi=400)

plt.figure(1)
plt.spy(F_Matrix)
plt.title('Spy plot of F')
plt.savefig('Spy_F.png', dpi=400)
```





0.1.3 Section III: Cre te Iter tive Schemes to Solve Diffusion Problem

[23]:
```
#---Generate Inverse Power Iteration Scheme---

def K_Converged(kguess,keff):
    tol = np.sqrt((1/num_cells) * ((keff-kguess)/keff)**2)
    return tol

def Phi_Converged(phiguess, phi):
    tol = np.sqrt((1/num_cells) * ((phi-phiguess)/phi)**2)
    return tol

tolerance = 1e-5

#Good to converge on both eigenvalue and vector for the modified IPI scheme,␣
   adds some computations but is well worth it

def IPI(phiguess, phinorm, kguess, D, F, num_cells):
    iterations = 0
    Phi_guess = phiguess / (sum(phiguess))
```

```
    RHS = (1/kguess) * np.dot(F, Phi_guess)
    kold = kguess
    phiold = Phi_guess
    phiold_sum = sum(phiold)
    D_inv = np.linalg.inv(D)
    Converged = F lse
    while Converged == F lse:
        iterations = iterations + 1
        Phi = np.dot(D_inv, RHS)
        Phi_sum = sum(Phi)
        keff = np.sum(np.dot(F,Phi)) / np.sum(np.dot(F,phiold)) * kold
        if K_Converged(kold,keff) <= tolerance  nd Phi_Converged(phiold_sum,␣
  Phi_sum) <= tolerance:
            Flux = Phi / (sum(Phi)) * phinorm
            K = keff
            RHS = RHS/(sum(RHS))
            Converged = True
            return Flux, K, iterations
        kold = keff
        phiold = Phi / sum(Phi) * phinorm
        phiold_sum = Phi_sum
        RHS = (1/kold)*np.dot(F,phiold)
```

[24]:
```
#---Utilize Power Iteration to Solve the 2D Diffusion Problem---

Phi= np.ones([n_groups*num_cells]) #Generate a guess of the flux vector␣
  instantiated to 1s
Phinorm = sum(Phi) #Initial normalized value
K_Guess = 1.02 #Guess K-effective

Phi_IPI,Keff_IPI,Iterations_IPI = IPI(Phi, Phinorm, K_Guess, D_Matrix,␣
  F_Matrix, num_cells)
```

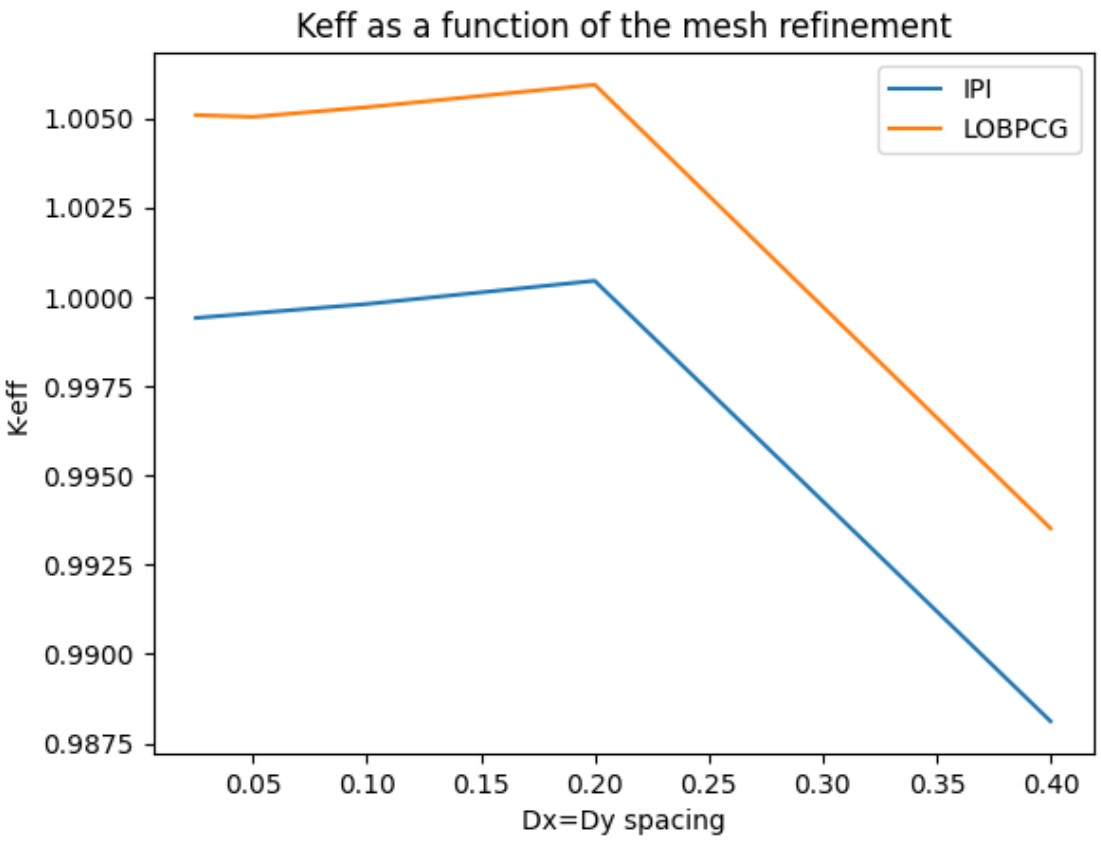
[25]:
```
#---Generate LOBPCG Iteration Scheme for k-eigenvalues---

#First - Define function to generate the generalized rayleigh quotient
def GeneralizedRQ( ,B,x):
    xBx = x.T.dot(B).dot(x)
    x x = x.T.dot( ).dot(x)
    RQ = xBx / x x
    return RQ

#Compute function to determine convergence on preconditioned residual
def LOBPCG_Conv_Test(w1, w2):
    Diff_Vec = w2 - w1
    epsilon = np.linalg.norm(Diff_Vec)
    return epsilon
```

```
#Compute function to quickly perform Rayleigh-Ritz method from three term␣
  reccurance subspace S
def Rayleigh_Ritz(Pencil, S):
    S_star = np.conjugate(S)
    S_star_T = np.transpose(S_star) #complex conjugate transpose
    RR_Matrix = S_star_T.dot(Pencil).dot(S)
    #solve RR eigenvalue problem
    ritz_vals, y = np.linalg.eig(RR_Matrix)
    ritz_vecs = S.dot(y)
    return ritz_vals, ritz_vecs

#LOBPCG!!!
def LOBPCG( ,B,phi,T,maxiter): #  matrix, B matrix, guess of eigenvectors, and␣
  preconditioner T with user specified max iterations
    if T is None:
        T = np.identity(np.size(phi))
    else:
        T = T
    if B is None:
        B = np.identity(np.size(phi))
    else:
        B = B
    if maxiter is None:
        maxiter = 100
    else:
        maxiter = maxiter
    converged = F lse
    iterations = 0
    tolerance = 1e-5
    p = np.zeros(np.size(phi))
    w_1 = 0
    while(converged == F lse):
        iterations = iterations + 1
        GRQ = GeneralizedRQ( ,B,phi)
        r = B.dot(phi) - (GRQ *  ).dot(phi)
        w_2 = T.dot(r)
        Pencil = B - (GRQ *  )
        TrialSubspace = np.column_stack((w_2,phi,p))
        vals,vecs = Rayleigh_Ritz(Pencil, TrialSubspace)
        phi_new = w_2 + vals[1]*phi + vals[2]*p
        p = w_2 + vals[2]*p
        epsilon = LOBPCG_Conv_Test(w_1, w_2)
        if epsilon <= tolerance or iterations >= maxiter:
            converged = True
            k_eff = GeneralizedRQ( ,B,phi_new)
            phi_new = phi_new / sum(phi_new)
```

```
            return phi_new, k_eff, iterations
        else:
            converged = F lse
            w_1 = w_2
            phi_new = phi_new / sum(phi_new)
            phi = phi_new
```

[26]:
```
#---Use LOBPCG to Solve the 2D Diffusion Problem---

Phi = np.ones([n_groups*num_cells]) #Generate a guess of the flux vector␣
  instantiated to 1s
Phi = Phi / np.sum(Phi)

Phi_LOBPCG,Keff_LOBPCG,Iterations_LOBPCG = LOBPCG(D_Matrix, F_Matrix, Phi,␣
  T=None, maxiter=None)
```

[27]:
```
#---Plot Mesh Refinement and K---

#This is of key interest - what is the convergent eigenvalue in small mesh sizes

#Values are obtianed by running the respective iterative method and setting the␣
  spacing value equal to the value in the Mesh_R array

K_IPI = [0.98812, 1.00045, 0.9998, 0.99954, 0.99941]
K_LOBPCG = [0.99352, 1.00594, 1.00531, 1.00504, 1.00509]
Mesh_R = [0.4,0.2,0.1,0.05,0.025] #refers to the dx=dy spacing in the mesh used

plt.figure(0)
plt.plot(Mesh_R,K_IPI)
plt.plot(Mesh_R, K_LOBPCG)
plt.title('Keff as a function of the mesh refinement')
plt.xlabel('Dx=Dy spacing')
plt.ylabel('K-eff')
plt.legend(['IPI', 'LOBPCG'])
plt.savefig('MeshSensitivity.png', dpi=400)
```


Keff as a function of the mesh refinement
IPI
LOBPCG
1.0050
1.0025
1.0000
0.9975
0.9950
0.9925
0.9900
0.9875
K-eff
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
Dx=Dy spacing