```
#%%

**Team 24 -- Quantum Sensor Lab -- Development of a Magnetic Field Steering System**

Andrew Fine - Project Manager \
Trevor Peterson - Logistics Manager \
Steven Coyan - System Engineer \
Nate Howard - Software Engineer \
Nick Elsasser - Software Engineer \
Aaron Li - Software Engineer \
Cole Sites - CAD Engineer/Finance Manager \
Dean Allison - Manufacturing Engineer \
Thomas Brewster - Test Engineer

---

Questions:

1) How do the coil numbers, sizes, positions, and currents interact to create a magnetic field and how can they be manipulated to change the field?

2) Can we simulate the magnetic field fast enough to test PID control loops in real time?

Ideas:

Implement a basic magnetic field simulation with some simplifying assumptions to get a baseline for how the main coil properties interact with one another to create a magnetic field and also to get a sense of how long it takes to query for the magnetic field at a given (x, y, z). Our simulation is based off of the equations from a paper provided to us by our client. We  also want to be able to take slices of the magnetic field to visualize how the field is created from the coils.

#%%

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from time import perf_counter_ns

#%%

#
# Simulation follows an implementation of:
#
```

```python
# Equations for the Magnetic Field Produced by One
# or More Rectangular Loops of Wire in the Same Plane - Misakian
#
# https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=10634
#
#

#
# Define helper functions
#

r1 = lambda x, y, z, a1, b1, z0: np.sqrt((a1 + x) ** 2 + (y + b1) ** 2 + (z - z0) ** 2)
r2 = lambda x, y, z, a1, b1, z0: np.sqrt((a1 - x) ** 2 + (y + b1) ** 2 + (z - z0) ** 2)
r3 = lambda x, y, z, a1, b1, z0: np.sqrt((a1 - x) ** 2 + (y - b1) ** 2 + (z - z0) ** 2)
r4 = lambda x, y, z, a1, b1, z0: np.sqrt((a1 + x) ** 2 + (y - b1) ** 2 + (z - z0) ** 2)

d1 = lambda x, y, z, a1, b1, z0: y + b1
d2 = lambda x, y, z, a1, b1, z0: y + b1
d3 = lambda x, y, z, a1, b1, z0: y - b1
d4 = lambda x, y, z, a1, b1, z0: y - b1

c1 = lambda x, y, z, a1, b1, z0: a1 + x
c2 = lambda x, y, z, a1, b1, z0: a1 - x
c3 = lambda x, y, z, a1, b1, z0: -a1 + x
c4 = lambda x, y, z, a1, b1, z0: -a1 - x

def _eval_r(x, y, z, a1, b1, z0):
    return (r1(x, y, z, a1, b1, z0), r2(x, y, z, a1, b1, z0),
            r3(x, y, z, a1, b1, z0), r4(x, y, z, a1, b1, z0))


def _eval_d(x, y, z, a1, b1, z0):
    return (d1(x, y, z, a1, b1, z0), d2(x, y, z, a1, b1, z0),
            d3(x, y, z, a1, b1, z0), d4(x, y, z, a1, b1, z0))


def _eval_c(x, y, z, a1, b1, z0):
    return (c1(x, y, z, a1, b1, z0), c2(x, y, z, a1, b1, z0),
            c3(x, y, z, a1, b1, z0), c4(x, y, z, a1, b1, z0
```

```python
))

#%%

# Magnetic permeability constant
MU_0 = 4 * np.pi * 1.0e-7

#
# Define field component equations, B = <Bx, By, Bz>
#

def Bx(x, y, z, a1, b1, z0, N):
    """
    Magnetic flux density in X direction at point (x, y, z) for a
    coil of size 2*a1, 2*b1 residing on the x, y plane defined by z0

    :param x: x location
    :param y: y location
    :param z: z location
    :param a1: (half of) coil side length A
    :param b1: (half of) coil side length B
    :param z0: Distance of coil from origin
    :param N: Current

    :return: Magnetic flux density
    """

    _k = 100.0 * MU_0 * N / (4 * np.pi)
    _z = z - z0
    _r = _eval_r(x, y, z, a1, b1, z0)
    _d = _eval_d(x, y, z, a1, b1, z0)

    return _k * np.sum([(-1**i) * _z / (_r[i] * (_r[i] + _d[i]))
                        for i in range(4)], axis=0)

def By(x, y, z, a1, b1, z0, N):
    """
    Magnetic flux density in Y direction at point (x, y, z) for a
    coil of size 2*a1, 2*b1 residing on the x, y plane defined by z0

    :param x: x location
    :param y: y location
    :param z: z location
    :param a1: (half of) coil side length A
    :param b1: (half of) coil side length B
```

```
    :param z0: Distance of coil from origin
    :param N: Current

    :return: Magnetic flux density
    """

    _k = 100.0 * MU_0 * N / (4 * np.pi)
    _z = z - z0

    _r = _eval_r(x, y, z, a1, b1, z0)
    _c = _eval_c(x, y, z, a1, b1, z0)

    return _k * np.sum([(-1**i) * _z / (_r[i] * (_r[i] + (-1**i) * _c[i]))
                        for i in range(4)], axis=0)

def Bz(x, y, z, a1, b1, z0, N):
    """
    Magnetic flux density in Z direction at point (x, y, z) for a
    coil of size 2*a1, 2*b1 residing on the x, y plane defined by z0

    :param x: x location
    :param y: y location
    :param z: z location
    :param a1: (half of) coil side length A
    :param b1: (half of) coil side length B
    :param z0: Distance of coil from origin
    :param N: Current

    :return: Magnetic flux density
    """

    _k = 100.0 * 1e6 * MU_0 * N / (4 * np.pi)

    # compute each sub-function once and store result
    _r = _eval_r(x, y, z, a1, b1, z0)
    _d = _eval_d(x, y, z, a1, b1, z0)
    _c = _eval_c(x, y, z, a1, b1, z0)

    # return constant * inner sum of various function combinations
    # tried to keep things clean, or at least cleaner than the original mathematica code

    return _k * np.sum([((-1**(i+1))*_d[i] / (_r[i] * (_r[i] + (-1**i)*_c[i]))) -
                        (_c[i] / (_r[i] * (_r[i] - _d[i]))) for i in range(4)], axis=0)
```

```python

#%%

def get_coil_coordinates(a1, b1, s, N, x, y):
    """
    Get the (x, y) coordinates for the center of the NxN grid of rectangles
    so they satisfy the distances set by the rectangle width, height, and spacing.
    The coils are positioned such that their combined center is at x, y

    :param a1: Rectangle width (x-direction) (cm)
    :param b1: Rectangle height (y-direction) (cm)
    :param s: Spacing between rectangles (cm)
    :param N: Number of rectangles
    :param x: x position of rectangle's center
    :param y: y position of rectangle's center

    :return: np.meshgrid in ij format of each coil's center for NxN grid
    """

    f = lambda x: (s * (N-1) + 2 * x * (N-1)) / 2.0

    f_a1, f_b1 = f(a1), f(b1)
    xx = np.linspace(-1.0 * f_a1, f_a1, num=N) + x
    yy = np.linspace(-1.0 * f_b1, f_b1, num=N) + y

    return np.meshgrid(xx, yy)

def get_field(x, y, z, a1, b1, z0, current, num_coils, coil_spacing):
    """
    Compute the magnetic field at (x, y, z) created by N^2 coils on plane z0 arranged in an NxN grid with equal spacing
    and current

    :param x: x coordinate to query (cm)
    :param y: y coordinate to query (cm)
    :param z: z coordinate to query (cm)
    :param a1: Half of the rectangle width (cm)
    :param b1: Half of the rectangle height (cm)
    :param z0: z coordinate plane goes through
    :param current: Current through all coils
    :param num_coils: Number of coils per row in grid, total coils is NxN
    :param coil_spacing: Spacing between coils (cm)

```

```
    :return: np.array containing the field x, y, z components in that order
    """
    xx, yy = get_coil_coordinates(a1, b1, coil_spacing, num_coils, x, y)

    bx, by, bz = None, None, None
    for i in range(num_coils):
        for j in range(num_coils):
            if bx is None:
                bx = Bx(xx[i, j], yy[i, j], z, a1, b1, z0, current)
            else:
                bx += Bx(xx[i, j], yy[i, j], z, a1, b1, z0, current)

            if by is None:
                by = By(xx[i, j], yy[i, j], z, a1, b1, z0, current)
            else:
                by += By(xx[i, j], yy[i, j], z, a1, b1, z0, current)

            if bz is None:
                bz = Bz(xx[i, j], yy[i, j], z, a1, b1, z0, current)
            else:
                bz += Bz(xx[i, j], yy[i, j], z, a1, b1, z0, current)

    return np.array([bx, by, bz])

def overlay_coils(a1, b1, num_coils, coil_spacing, unit, ax):
    """
    Overlay where the coils would be on the x, y plane in the magnetic field plot

    :param a1: half of the coil width (cm)
    :param b1: half of the coil height (cm)
    :param num_coils: Number of coils per row
    :param coil_spacing: Spacing between coils (cm)
    :param unit: Conversion between cm and matplotlib pixel units
    :param ax: matplotlib graph to apply overlay to
    """

    # convert coil sizes to matplotlib units
    a1 *= unit
    b1 *= unit
```

```
    coil_spacing *= unit

    f = lambda x: (coil_spacing * (num_coils - 1) + 2 * x * (num_coils - 1)) / 2.0

    f_a1, f_b1 = f(a1), f(b1)

    # shift up and over s.t. we have the top left corner of each coil
    xp = np.linspace(-1.0 * f_a1 + 50, f_a1 + 50, num=num_coils) - a1
    yp = np.linspace(-1.0 * f_b1 + 50, f_b1 + 50, num=num_coils) - b1
    xx, yy = np.meshgrid(xp, yp)
    for i in range(num_coils):
        for j in range(num_coils):
            px, py = xx[i, j], yy[i, j]
            print(px, py, 2*a1, 2*b1)
            rect = plt.Rectangle((px, py), 2*a1, 2*b1, fc='none', ec='green', lw=2.0, alpha=0.75)
            ax.add_patch(rect)



def overlay_coilsY(a1, b1, num_coils, coil_spacing, unit, ax):
    # convert coil sizes to matplotlib units
    a1 *= unit
    b1 *= unit
    coil_spacing *= unit

    f = lambda x: (coil_spacing * (num_coils - 1) + 2 * x * (num_coils - 1)) / 2.0

    f_a1, f_b1 = f(a1), f(b1)

    # shift up and over s.t. we have the top left corner of each coil
    xp = np.linspace(-1.0 * f_a1 + 50, f_a1 + 50, num=num_coils) - a1
    yp = np.linspace(-1.0 * f_b1 + 50, f_b1 + 50, num=num_coils) - b1
    xx, yy = np.meshgrid(xp, yp)
    for i in range(num_coils):
        for j in range(num_coils):
            if i==1:
                continue
            px, py = xx[i, j], yy[i, j]
            #print(px, py, 2*a1, 2*b1)
            rect = plt.Rectangle((px, py+(2*unit)),2*a1,.5
```

```
, fc='none', ec='green', lw=2.0, alpha=0.75)
            ax.add_patch(rect)

def overlay_coilsX(a1, b1, num_coils, coil_spacing, unit, ax):
    # convert coil sizes to matplotlib units
    a1 *= unit
    b1 *= unit
    coil_spacing *= unit

    f = lambda x: (coil_spacing * (num_coils - 1) + 2 * x * (num_coils - 1)) / 2.0

    f_a1, f_b1 = f(a1), f(b1)

    # shift up and over s.t. we have the top left corner of each coil
    xp = np.linspace(-1.0 * f_a1 + 50, f_a1 + 50, num=num_coils) - a1
    yp = np.linspace(-1.0 * f_b1 + 50, f_b1 + 50, num=num_coils) - b1
    xx, yy = np.meshgrid(xp, yp)
    for i in range(num_coils):
        for j in range(num_coils):
            if j==1:
                continue
            px, py = xx[i, j], yy[i, j]
            #print(px, py, 2*a1, 2*b1)
            rect = plt.Rectangle((px+(2*unit), py), .5, 2*b1, fc='none', ec='green', lw=2.0, alpha=0.75)
            ax.add_patch(rect)

def create_field(const, val):
    # define some random constants to plot
    x_axis = np.linspace(-10, 10, num=100)
    y_axis = np.linspace(-10, 10, num=100)

    # A1 and B1 are (half of) the rectangle length and width
    A1, B1, I, spacing = 2, 2, 1, 1.0

    z_wall_1 = -100.0 # 100 cm back
    z_wall_2 = 100.0 # 100 cm forward

    GRID = 3
    xx, yy = get_coil_coordinates(A1, B1, spacing, GRID, 0, 0)
    size = max((GRID-1), 1) * A1 * spacing * 2

    grid = np.zeros((100, 100))
```

```

    axis1 = np.linspace(-size, size, 100)
    axis2 = np.linspace(-size, size, 100)

    unit = 100 / (2*size) # convert cm grid (2*size x 2*size) to matplotlib unit grid (100x100 plot)
    for i, ax1 in enumerate(axis1):
        for j, ax2 in enumerate(axis2):
            # for k, z in ...
            # visualize coil placement by plotting distance to closest coil for each (x,y)

            # v = [np.sqrt((x - xx[ii, jj])**2 + (y - yy[ii, jj])**2)
            #     for ii in range(GRID) for jj in range(GRID)]
            # grid[i, j] = min(v)

            # get the magnitude of the field at each x, y
            if const == 'z':
                B_wall_1 = get_field(ax1, ax2, val, A1, B1, z_wall_1, I, GRID, spacing)
                B_wall_2 = get_field(ax1, ax2, val, A1, B1, z_wall_2, I, GRID, spacing)
            elif const == 'y':
                B_wall_1 = get_field(ax1, val, ax2, A1, B1, z_wall_1, I, GRID, spacing)
                B_wall_2 = get_field(ax1, val, ax2, A1, B1, z_wall_2, I, GRID, spacing)
            elif const =='x':
                B_wall_1 = get_field(val, ax1, ax2, A1, B1, z_wall_1, I, GRID, spacing)
                B_wall_2 = get_field(val, ax1, ax2, A1, B1, z_wall_2, I, GRID, spacing)

            B = B_wall_1 + B_wall_2

            u, v, w = B[0], B[1], B[2]

            grid[i, j] = np.linalg.norm(B, axis=0)
    # plot the grid
    xticks = ['' if i % 10 else f'{x_axis[i]:.2f}' for i in range(len(axis1))][:-1] + [axis1[-1]]
    yticks = ['' if i % 10 else f'{y_axis[i]:.2f}' for i in range(len(axis2))][:-1] + [axis2[-1]]
    ax = sns.heatmap(grid, xticklabels=xticks, yticklabels=yticks)

    if const == 'z':
        overlay_coils(A1, B1, GRID, spacing, unit, ax)
```

```python

        plt.title(f'Magnetic Field Magnitude (nT) of a {GRID}x{GRID} of Coils at Z={val}cm.')
        plt.xlabel('X position (cm)')
        plt.ylabel('Y position (cm)')

        plt.show()
    elif const == 'y':
        overlay_coilsY(A1, B1, GRID, spacing, unit, ax)

        plt.title(f'Magnetic Field Magnitude (nT) of a {GRID}x{GRID} of Coils at Y={val}cm.')
        plt.xlabel('X position (cm)')
        plt.ylabel('Z position (cm)')

        plt.show()
    elif const == 'x':
        overlay_coilsX(A1, B1, GRID, spacing, unit, ax)

        plt.title(f'Magnetic Field Magnitude (nT) of a {GRID}x{GRID} of Coils at X={val}cm.')
        plt.xlabel('Y position (cm)')
        plt.ylabel('Z position (cm)')

        plt.show()

#%%

# Define parameters for our simulation that are similar to our actual parameters (so far)

# wall size
wall_size = 160.0 # cm

# number of coils
N = 4

spacing = 10.0 # cm

# square coils, 4x4 grid taking up 1.6m x 1.6m, 160 / 4 - spacing * 3 =
a1 = (wall_size / N) - (spacing * (N-1)) # cm
b1 = a1 # cm

# 1.4m of separation, let the origin be directly at the center
sep = 140.0 # cm
z_wall_1 = sep / 2.0 # cm
z_wall_2 = -z_wall_1 # cm

```

```python
# for now have the current be exactly the same for all coils
I = 1.0 # Amps

# time querying field between the walls at 0, 0, 0
s = perf_counter_ns()
get_field(0, 0, 0, a1, b1, z_wall_1, I, N, spacing) + get_field(0, 0, 0, a1, b1, z_wall_2, I, N, spacing)
e = perf_counter_ns() - s
print(f'Time to query field: {e/1e6:.2f} milliseconds')

```

```python
#%%

create_field('x',1)
create_field('y',1)
create_field('z',1)

```

```
#%% md
```

Insights gained from prototype:

1) With a simulation we are easily able to see how the variable coil parameters interact with each other to create the magnetic field. As our design changes we can change our simulation to follow and quickly see how any changes would impact the overall performance of magnetic field generation.

2) We also see that our simulation provides a good first step to simulating and testing our PID loops as it only takes a couple milliseconds to query one point in the field. We are also able to visualize our magnetic fields and coils through our simulation. Given better parallization of the code we may be able to speed up the visualizations to near real time, and we are also looking into full 3d visualizations of the coils and magnetic field as a next step for the simulation.