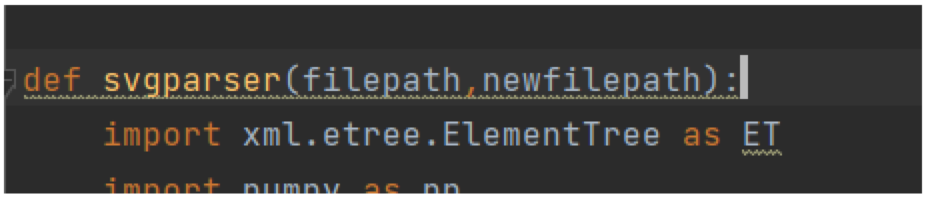
Documentation:

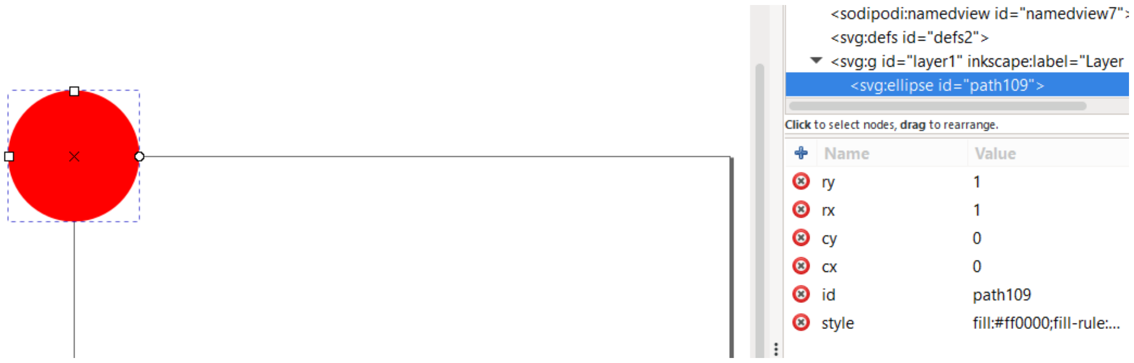
Note: All the values produced/entered by the user are in S.I. units e.g. meters are used instead of millimeters etc.

Drawing in Inkscape:
The purpose of this function is to parse an SVG file that is created in Inkscape and convert it into an Excel file that is more useful to us. One thing to note however is that for an unconnected path with more than 3 sides, the program automatically attempts to connect the start and the end nodes together to form a polygon, this function can be turned off in the program itself, but it is done because the file produced by inkscape doesn't explicitly say if the paths itself are connected or not.



```
def svgparser(filepath,newfilepath):
    import xml.etree.ElementTree as ET
    import numpy as np
```

Inkscape:

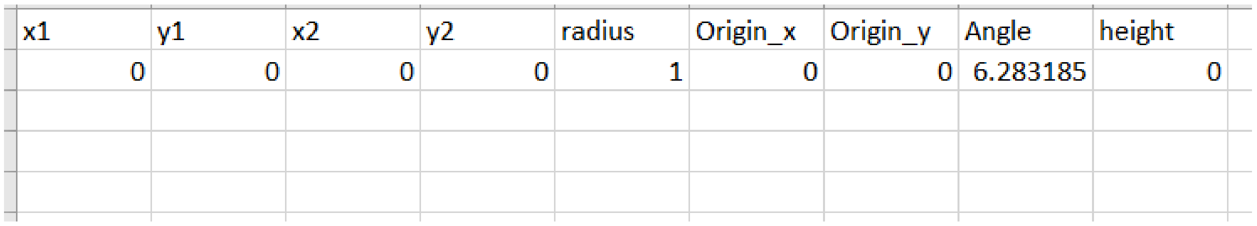


```
<sodipodi:namedview id="namedview7">
<svg:defs id="defs2">
<svg:g id="layer1" inkscape:label="Layer
  <svg:ellipse id="path109">
```

Click to select nodes, drag to rearrange.

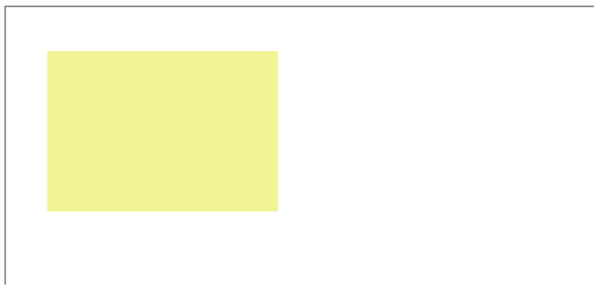
Name	Value
ry	1
rx	1
cy	0
cx	0
id	path109
style	fill:#ff0000;fill-rule:...

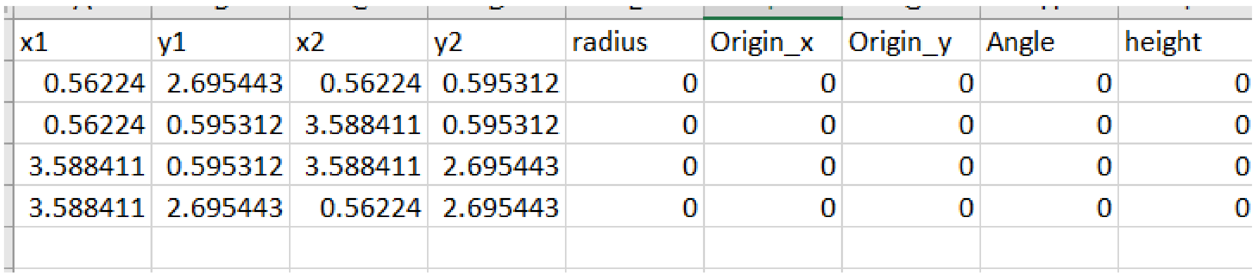
Excel:



x1	y1	x2	y2	radius	Origin_x	Origin_y	Angle	height
0	0	0	0	1	0	0	6.283185	0

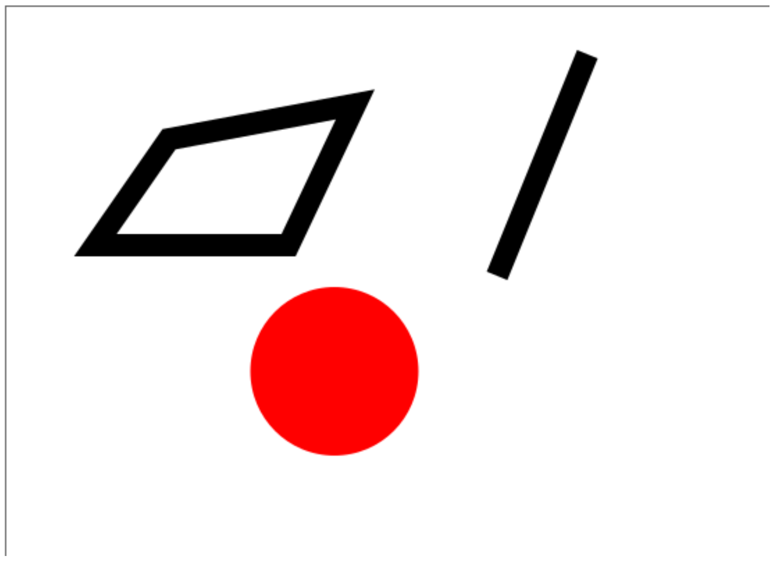
Examples of different shapes.



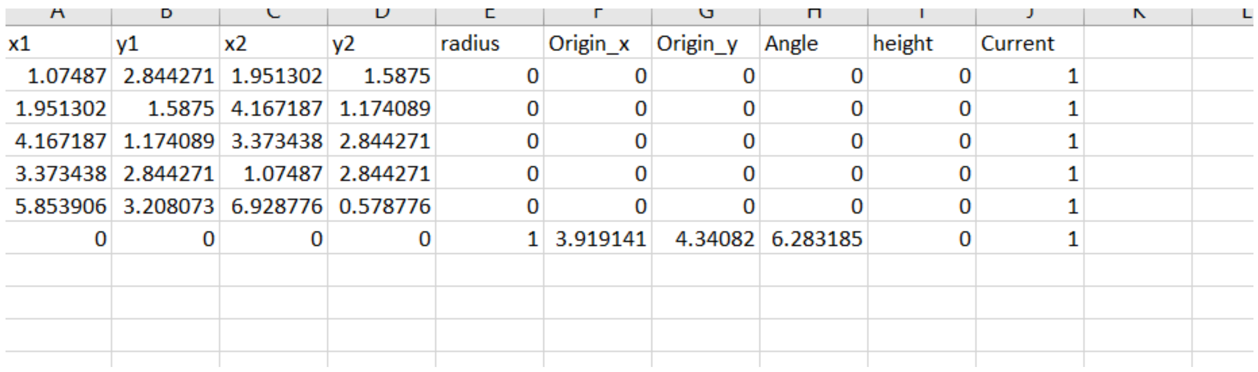


x1	y1	x2	y2	radius	Origin_x	Origin_y	Angle	height
0.56224	2.695443	0.56224	0.595312	0	0	0	0	0
0.56224	0.595312	3.588411	0.595312	0	0	0	0	0
3.588411	0.595312	3.588411	2.695443	0	0	0	0	0
3.588411	2.695443	0.56224	2.695443	0	0	0	0	0

Example of arbitrary groups of shapes being parsed by the function, i.e. how the parser handles, an image containing some line, a path and a circle/arc.



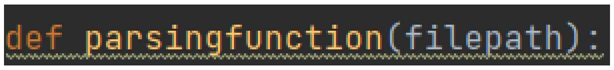
Parsed:



x1	y1	x2	y2	radius	Origin_x	Origin_y	Angle	height	Current
1.07487	2.844271	1.951302	1.5875	0	0	0	0	0	1
1.951302	1.5875	4.167187	1.174089	0	0	0	0	0	1
4.167187	1.174089	3.373438	2.844271	0	0	0	0	0	1
3.373438	2.844271	1.07487	2.844271	0	0	0	0	0	1
5.853906	3.208073	6.928776	0.578776	0	0	0	0	0	1
0	0	0	0	1	3.919141	4.34082	6.283185	0	1

Do note, that the current and height options need to be entered manually by going into the file by the user, as there is no support for that in Inkscape. The current is set at a default of 1 Amps and height is set to a default at 0. This is purely for future development and should be left as 1 by the user and the current through the wire needs to be changed in the function. The height can be manipulated for ONLY circles and arcs, from which the user can create a "pseudo-solenoid" by stacking loops of wire on top of each other. 3D functionality around a line can be added later by adding a z1 and z2 column, however as to how this would be able to be drawn in Inkscape needs to be investigated further.

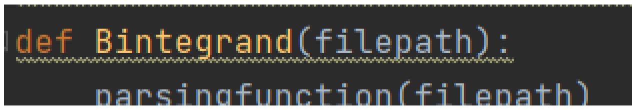
Parsing function:



```
def parsingfunction(filepath):
```

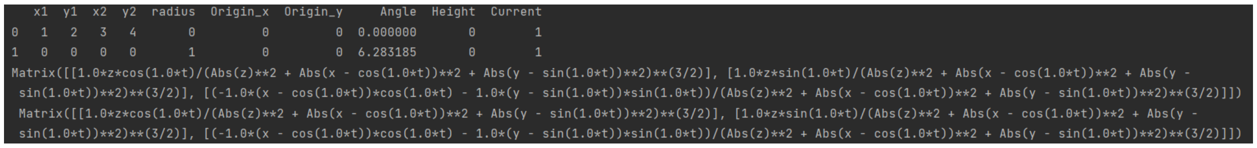
Essentially it takes the excel file and stores it as temporary arrays, used in another function to find the Integrand of the B-field.

Bintegrand:



```
def Bintegrand(filepath):
    parsingfunction(filepath)
```

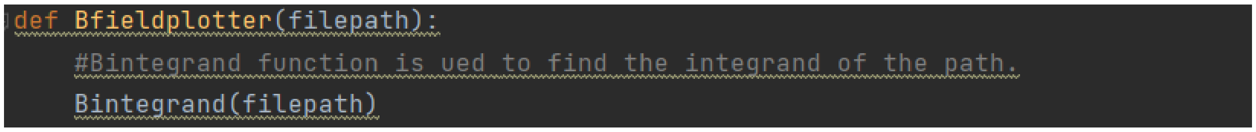
The user needs to enter the filepath of the spreadsheet.



```
   x1  y1  x2  y2  radius  Origin_x  Origin_y     Angle  Height  Current
0   1   2   3   4       0         0         0  0.000000       0        1
1   0   0   0   0       1         0         0  6.283185       0        1
Matrix([[1.0*z*cos(1.0*t)/(Abs(z)**2 + Abs(x - cos(1.0*t))**2 + Abs(y - sin(1.0*t))**2)**(3/2)], [1.0*z*sin(1.0*t)/(Abs(z)**2 + Abs(x - cos(1.0*t))**2 + Abs(y -
 sin(1.0*t))**2)**(3/2)], [(-1.0*(x - cos(1.0*t))*cos(1.0*t) - 1.0*(y - sin(1.0*t))*sin(1.0*t))/(Abs(z)**2 + Abs(x - cos(1.0*t))**2 + Abs(y - sin(1.0*t))**2)**(3/2)]])
 Matrix([[1.0*z*cos(1.0*t)/(Abs(z)**2 + Abs(x - cos(1.0*t))**2 + Abs(y - sin(1.0*t))**2)**(3/2)], [1.0*z*sin(1.0*t)/(Abs(z)**2 + Abs(x - cos(1.0*t))**2 + Abs(y -
 sin(1.0*t))**2)**(3/2)], [(-1.0*(x - cos(1.0*t))*cos(1.0*t) - 1.0*(y - sin(1.0*t))*sin(1.0*t))/(Abs(z)**2 + Abs(x - cos(1.0*t))**2 + Abs(y - sin(1.0*t))**2)**(3/2)]])
```

The integrand is displayed to the user, it can be noted/plugged into another function/programme. The limits of integration for the integrand are 0 to 2*pi for an arc/circle and 0 to 1 for all line segments.
For a shape such as a rectangle, each line segment has its own integrand and this is superposed at a point to find the total B-field.

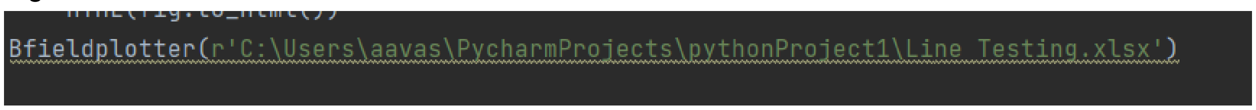
Bfiledplotter



```
def Bfieldplotter(filepath):
    #Bintegrand function is ued to find the integrand of the path.
    Bintegrand(filepath)
```

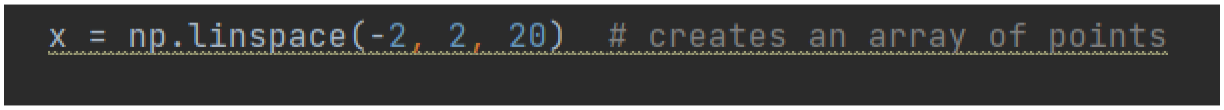
The purpose of this function is to take the excel spreadsheet and plot the magnetic field using cones to represent vectors, the user just needs to input the file path.

E.g.

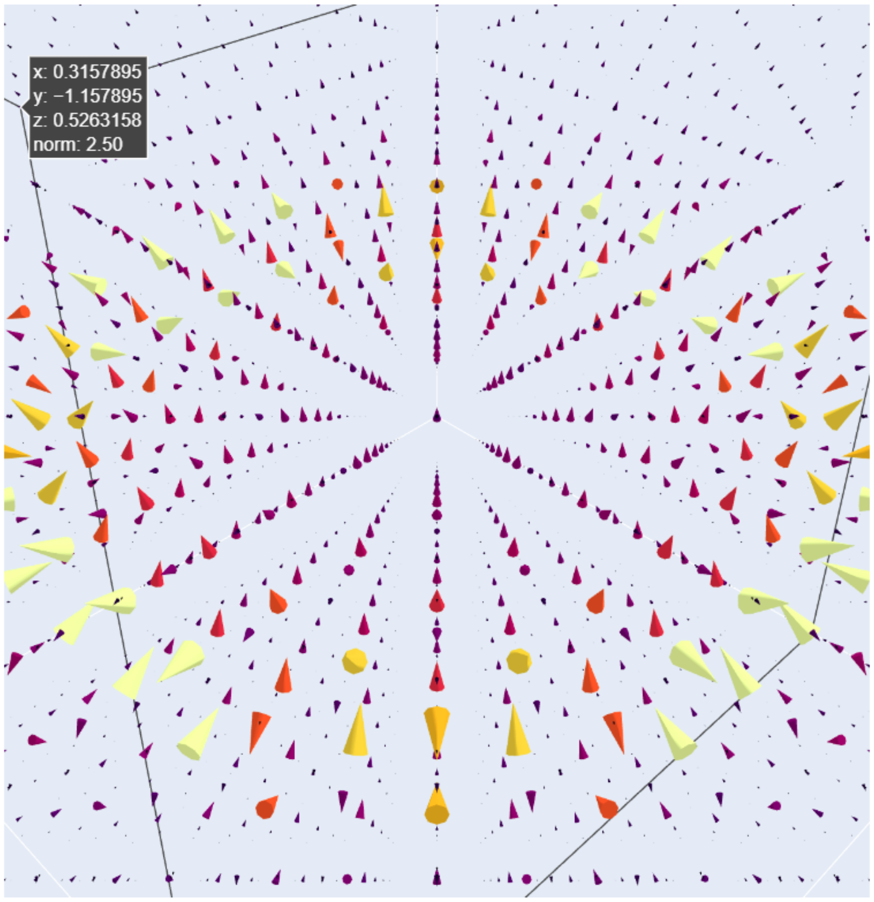


```
Bfieldplotter(r'C:\Users\aavas\PycharmProjects\pythonProject1\Line Testing.xlsx')
```

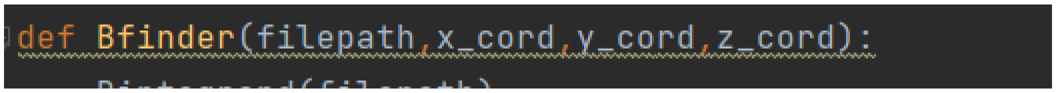
Would result in this HTML file being produced in the users browser, intractable. The range of values that the plot is valid for can be changed by the user within the function,



```
x = np.linspace(-2, 2, 20)  # creates an array of points
```

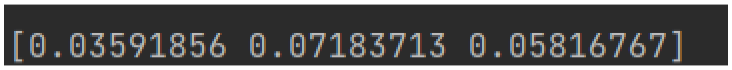


Bfinder



```
def Bfinder(filepath,x_cord,y_cord,z_cord):
```

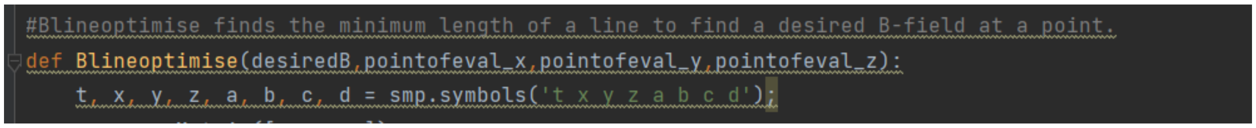
To find the B-field at a certain point, the user needs to input the desired point of evaluation and the path containing the excel files. Leads to an output of,



```
[0.03591856 0.07183713 0.05816767]
```

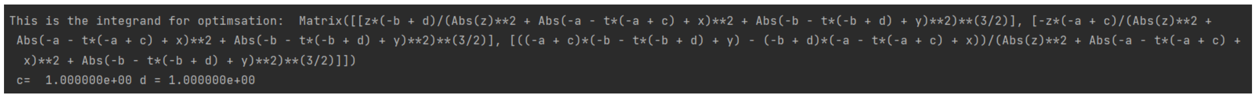
which is the B-field at a point.

Blineoptimise



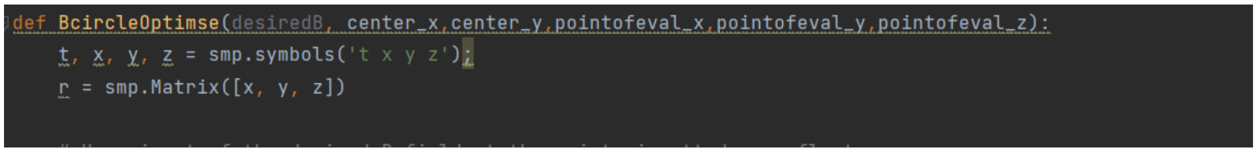
```
#Blineoptimise finds the minimum length of a line to find a desired B-field at a point.
def Blineoptimise(desiredB,pointofeval_x,pointofeval_y,pointofeval_z):
    t, x, y, z, a, b, c, d = smp.symbols('t x y z a b c d');
```

The Blineoptimise function is used to find the length of wire that is required to achieve a desired B-field at a point (x,y,z). The program sets one of the points to be at the origin (0,0) and it finds (c,d) which is the x,y coordinate of the end point. A limit is set to (1,1) so the program checks points between (0,0) to (1,1) to find a point in space that leads to a magnetic field, closest to the desired one at the point of evaluation. The limits can be changed and the accuracy can also be changed as desired. Output:



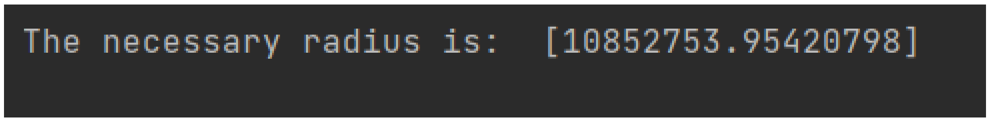
```
This is the integrand for optimsation:  Matrix([[z*(-b + d)/(Abs(z)**2 + Abs(-a - t*(-a + c) + x)**2 + Abs(-b - t*(-b + d) + y)**2)**(3/2)], [-z*(-a + c)/(Abs(z)**2 +
 Abs(-a - t*(-a + c) + x)**2 + Abs(-b - t*(-b + d) + y)**2)**(3/2)], [((-a + c)*(-b - t*(-b + d) + y) - (-b + d)*(-a - t*(-a + c) + x))/(Abs(z)**2 + Abs(-a - t*(-a + c) +
  x)**2 + Abs(-b - t*(-b + d) + y)**2)**(3/2)]])
 c=  1.000000e+00 d = 1.000000e+00
```

BcircleOptimise:



```
def BcircleOptimise(desiredB, center_x,center_y,pointofeval_x,pointofeval_y,pointofeval_z):
    t, x, y, z = smp.symbols('t x y z');
    r = smp.Matrix([x, y, z])
```

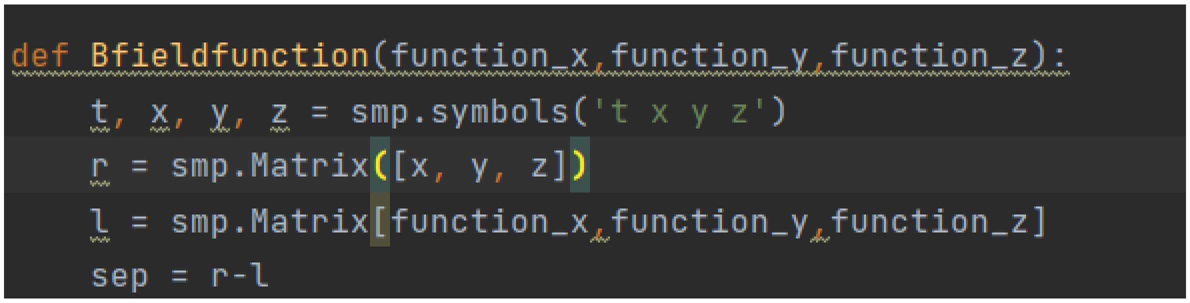
The purpose of this function allows the user to find the radius of a loop that is required to produce a desired B-field at a certain point of evaluation. The program produces a result of,



```
The necessary radius is:  [10852753.95420798]
```

WIP:

Bfieldfunction



```
def Bfieldfunction(function_x,function_y,function_z):
    t, x, y, z = smp.symbols('t x y z')
    r = smp.Matrix([x, y, z])
    l = smp.Matrix[function_x,function_y,function_z]
    sep = r-l
```

The purpose of this function is to find the B-field created by a parameterised function with known boundaries. It will use the symbol “t” and integrate over dt. The current problem with this function is that it works when it is manually inputted outside of a function but not quite working as a function itself. Can add further functionality for other variables or multiple variables to integrate over.

SelfLcalculator:

This function would be used to calculate the self inductance of a field, it is fairly simple to do so for a B-field generated by a solenoid consisting of a circular wire of loop as the B-field inside the solenoid is constant. However for more general shapes, a small area dA would need to be evaluated and the B-field of that point needs to be calculated. I will look further into this. (Potentially use numerical methods of integration, can also look into divs/grads/curls).

General Conclusions and findings for symbolic integration:

The major attraction for using the symbolic toolbox is that the output it produces can be passed along to another function, e.g. you can attach a linear optimization algorithm to the integrand produced. However one of the major drawbacks for this specific case is that, integrating symbolically takes a lot of computing power and time. It is not much of an issue if it was running on good hardware but benchmarked on a Ryzen 7, the integration takes approximately 10-15minutes per integral for each line segment, so this would indeed grow for more complex shapes. So a numerical method is used using the quad function in the numpy library. In addition to this, for a magnetic field calculation, using the quad approach leads to a percentage error of 2.0e-07, when compared to analytical results for the magnetic field at a point along the central axis of a circular loop, so the methodology is sufficient. To progress this package would mean to add more shapes to the parser and the ability to integrate over a loop of wire that has a non-constant radius, i.e. has a dR element too. Another functionality to add would be a Self/Mutual inductance calculator for the hand drawn shapes and this would probably be achieved by using Stokes theorem but would need to be investigated further. Furthermore, one feature that can be developed is to see how a 3D object (point-like or not) would behave within the magnetic field, and be able to see how the “reflected” magnetic field would be shaped, this would be done using dipole-dipole approximations.