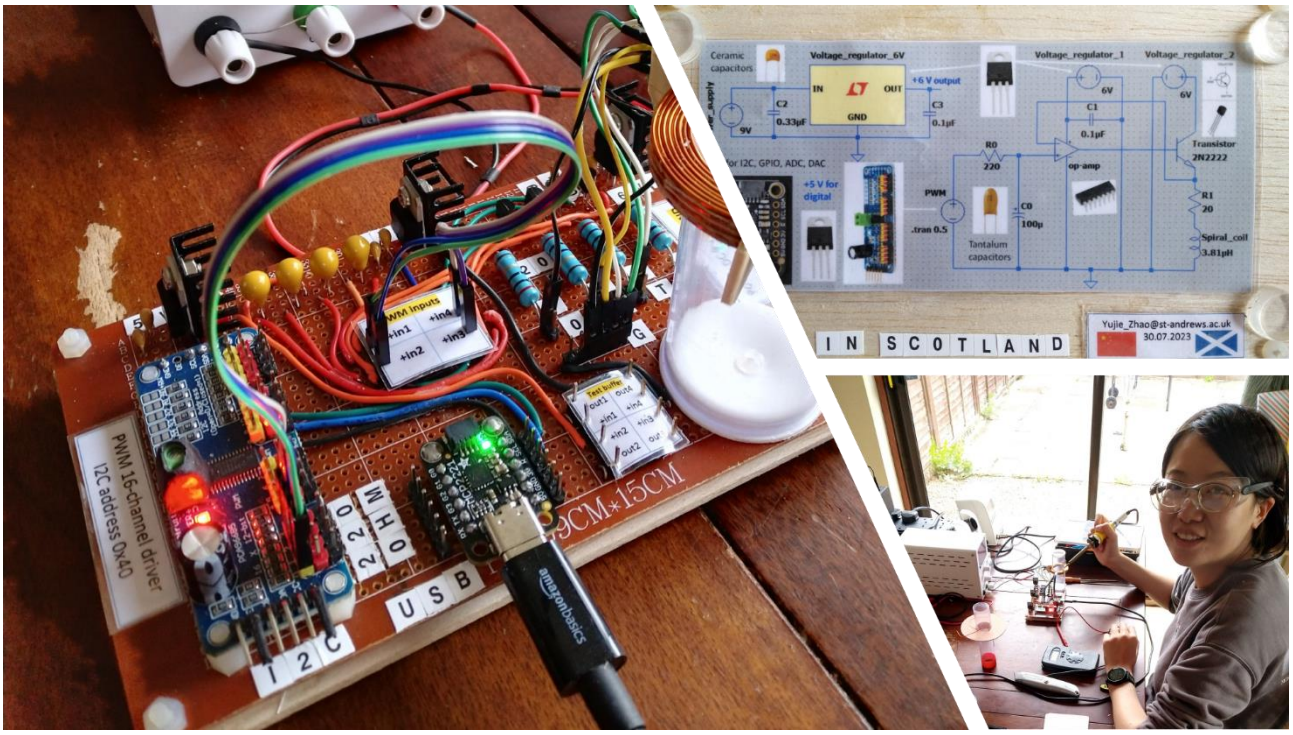
Design and optimisation of 1D shimming coils



Contents

1. Introduction

Existing commercial NMR and EPR systems achieve sample spin polarisation for producing resonance absorption by using external superconducting magnets. These magnets usually demonstrate a good homogeneity of magnetic field within a small volume. However, in the case of larger sample volumes, the homogeneity of the field may be compromised, leading to a non-coherent spin response. As a result, only the central region of the sample within a sufficiently homogeneous field will contribute to the detected RF response. To improve the signal-to-noise ratio, it is necessary to extend the sample volume while maintaining magnetic field homogeneity, which can increase the coherency of the response. In general, this problem can be addressed through the use of a supplementary coil system that is specifically designed to mitigate any field irregularities in the targeted volume.

The need to compensate for residual magnetic fields first emerged in the Navy during the 19th century. With the increasing use of metal parts in sailing fleets, compass readings were significantly distorted. Over time, a solution was proposed: as the compass needle aligns with the maximum field, any azimuthal distribution of a monotonically changing residual field becomes unimportant and can thus be compensated for in only one direction. The history of this problem and its elegant solution can be found on the Internet, such as in Wikipedia: *In 1854, a new type of binnacle was patented by John Gray of Liverpool which directly incorporated adjustable correcting magnets on screws or rack and pinions. This was improved again when Lord Kelvin patented in the 1880s another system of compass and which incorporated two compensating spheres. These are colloquially known as "Kelvin's balls" in the UK, and "navigator's balls" in the United States.* An example of Kelvin's binnacle with compensating balls, constituting a magnetic dipole, is shown in Fig. 1.

Fig. 1. An example of Kelvin's binnacle with magnetically compensating balls. RRS Discovery, Dundee, Scotland. Photographed by Yujie Zhao.

As depicted in Fig. 1, it is possible to compensate for field inhomogeneities by introducing supplementary field sources. In this report, we have developed two 1D coil optimisation algorithm that is capable of minimizing a residual field profile along a specified axis. Multidimensional residual field profiles will, of course, require more complex compensation systems where the coils can be placed on specific surfaces.

Another noteworthy application of shimming coils is their use in compensating for Earth's magnetic field, in addition to other sources, during the calibration of magnetic sensors. Fig. 2 illustrates a system (Aichi Steel, Japan) featuring three intersecting Helmholtz coils capable of compensating for a three-dimensional magnetic field vector. Compensating for a uniform field can only be achieved within a small volume at the centre. To address gradient fields, a greater number of coils and an optimization algorithm are necessary. Using this system as an example, there arises the necessity to develop electronic coil drivers.

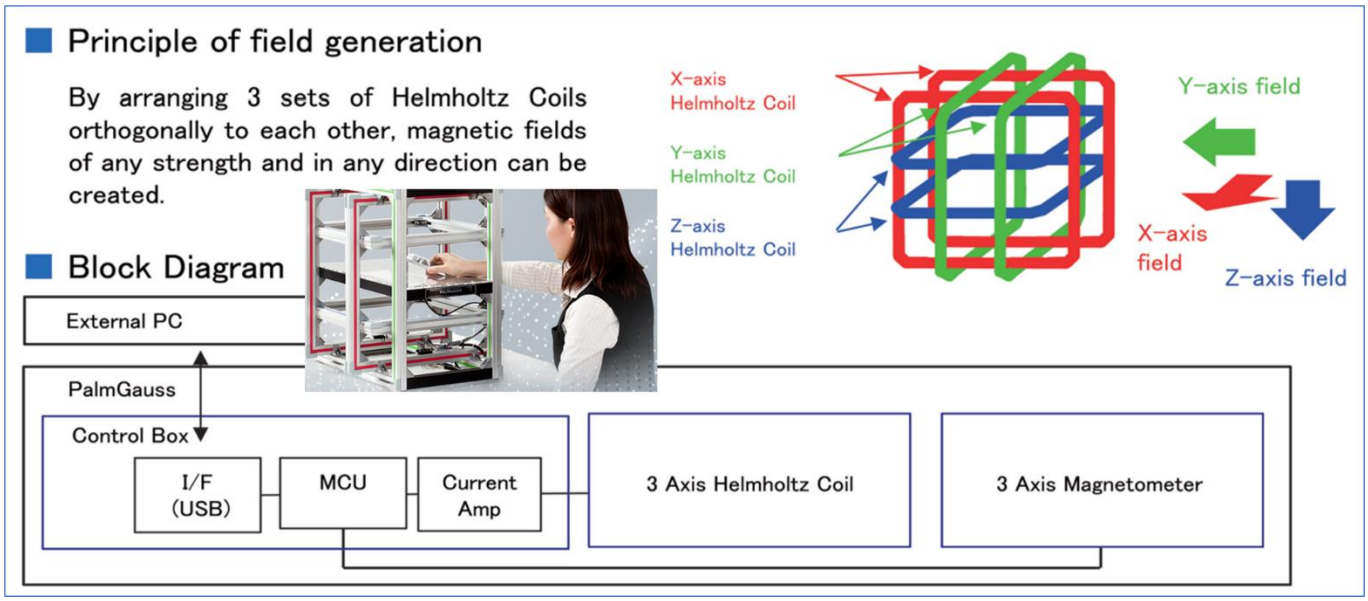


Fig. 2. An installation (Aichi Steel, Japan) comprising three intersecting Helmholtz coils, designed to compensate for the Earth's magnetic field and other external sources, is employed for the calibration of magnetic sensors.

In the project, our primary goal was not to develop a practical compensation system but to explore the mathematical methods, electronic and digital technologies required for future customized systems in our measurement installations. Along this path, we have made significant progress:

- Proposed a shimming coil architecture using a planar spiral coil system.
- Tested Python algorithms for numerical optimization of functionals with a large number of variables.
- Explored the Python symbolic calculus library SymPy, enabling complex calculations like derivatives.
- Developed and tested a low-cost multi-channel PWM driver circuit with a current booster, facilitating the powering of numerous individual coils.

- Presented a scheme for testing electronic drivers using the Arduino microcontroller.
- Tested a USB-to-I2C adapter, allowing direct computer control of multi-channel PWM drivers without the need for microcontrollers.
- Demonstrated effective heat management of electronic devices using voltage regulators and transistors.
- Identified potential manufacturing technologies for planar spiral coils.
- Successfully manufactured a working prototype of a four-channel electronic driver.

2. Planar spiral coil

The planar spiral coil will serve as the main element in the numerical optimisation algorithm. Fig. 3 shows the geometrical model of a planar spiral coil. The planar spiral coil is defined by the rotating radius vector $\vec{\mathbf{r}}_{d\theta}$ that points to the centre of the wire with a diameter of d. The minimum radius of the spiral is $d/2$. As the angle of rotation increases by θ, the displacement along the radius is given by $\gamma\theta$, where $\gamma \geq d/(2\pi)$ is the interturn parameter that synchronizes the radial displacement with the rotation:

$$\vec{\mathbf{r}}_{d\theta} = \begin{pmatrix} (\gamma\theta + d/2)\cos(\theta) \\ (\gamma\theta + d/2)\sin(\theta) \\ 0 \end{pmatrix} \quad (1)$$

In Eq. (1), the z-axis was selected to be perpendicular to the xy-plane, which is where the vector projections are calculated. In Fig. 3, the rotation is conducted in the positive counterclockwise direction, with the counter consistently positioned on the left side. A positive angle is used to indicate the magnitude of this rotation. Moreover, the positive direction of the z-axis is oriented towards the observer.

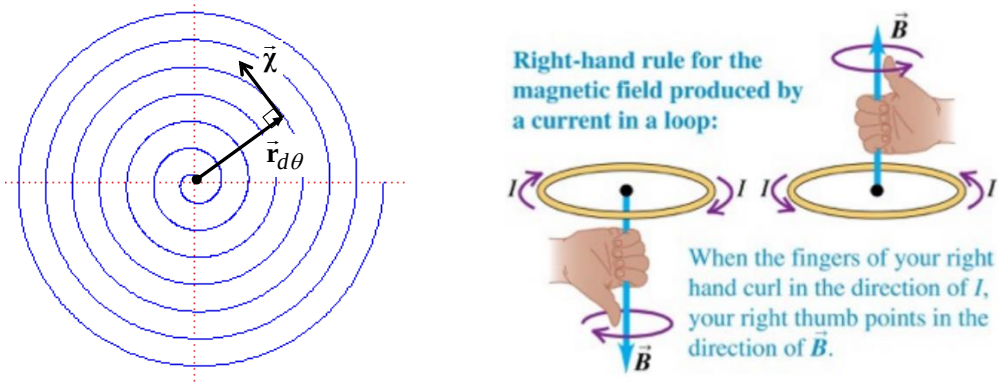


Fig. 3. Geometrical model of a spiral coil and method for determining the induction direction. It is assumed that the current follows the conventional direction, flowing from a positive terminal to a negative terminal.

In the SI system of physical units, the magnetic induction $\vec{\mathbf{B}}$ (measured in Tesla, T) resulting from a current in a thin wire can be calculated using the following contour integral:

$$\vec{\mathbf{B}} = \frac{\mu_0}{4\pi}\int_L \frac{[\vec{\mathbf{I}}\times\vec{\mathbf{r}}]dl}{|\vec{\mathbf{r}}|^3} = \frac{\mu_0}{4\pi} I \int_L \frac{[\vec{\boldsymbol{\chi}}\times\vec{\mathbf{r}}]dl}{|\vec{\mathbf{r}}|^3} \tag{2}$$

where $\mu_0 = 4\pi \times 10^{-7}$ (H/m, Henry per meter) is the vacuum magnetic constant, $[\vec{\mathbf{I}} \times \vec{\mathbf{r}}]$ and $[\vec{\boldsymbol{\chi}} \times \vec{\mathbf{r}}]$ are the vector product, I (A, Amperes) is the current magnitude (positive or negative) along the wire, $\vec{\boldsymbol{\chi}}$ is the positive unit tangent vector to the counter L in the integration point (see Fig. 3), and $\vec{\mathbf{r}}$ is the vector directed from the integration point to an observation point $\vec{\mathbf{r}}_0$. It is assumed that the current follows the conventional direction, flowing from a positive terminal to a negative terminal. If the direction of the current aligns with $\vec{\boldsymbol{\chi}}$ in Fig. 3, I will be considered positive.

Eq. (2) can be rewritten in terms of integration over the rotation angle:

$$\vec{\mathbf{B}}(\vec{\mathbf{r}}_0) = \frac{\mu_0 I}{4\pi}\int_L \frac{[\vec{\boldsymbol{\chi}}\times\vec{\mathbf{r}}]dl}{|\vec{\mathbf{r}}|^3} = \frac{\mu_0 I}{4\pi}\int_{\theta_1}^{\theta_2} \frac{[\vec{\mathbf{v}}\times\vec{\mathbf{r}}]}{|\vec{\mathbf{r}}|^3} d\theta \tag{3}$$

where the angles can be either positive or negative multiturn angles, indicating clockwise or anticlockwise spirals. The vector and scalar quantities in Eq. (3) can be expressed in terms of the radius vector:

$$\begin{cases} \vec{\boldsymbol{\chi}} = \frac{d\vec{\mathbf{r}}_{d\theta}}{d\theta} \times \frac{1}{\left|\frac{d\vec{\mathbf{r}}_{d\theta}}{d\theta}\right|} = \frac{\vec{\mathbf{v}}}{\sqrt{\gamma^2+(\gamma\theta+d/2)^2}} \\ \vec{\mathbf{v}} = \begin{pmatrix} \gamma\cos(\theta) - (\gamma\theta + d/2)\sin(\theta) \\ \gamma\sin(\theta) + (\gamma\theta + d/2)\cos(\theta) \\ 0 \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \\ dl = d\theta \left|\frac{d\vec{\mathbf{r}}_{d\theta}}{d\theta}\right| = d\theta\sqrt{\gamma^2 + (\gamma\theta + d/2)^2} \\ \vec{\mathbf{r}} = \vec{\mathbf{r}}_0 - \vec{\mathbf{r}}_{d\theta} = \begin{pmatrix} x_0 - (\gamma\theta + d/2)\cos(\theta) \\ y_0 - (\gamma\theta + d/2)\sin(\theta) \\ z_0 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \\ [\vec{\mathbf{v}} \times \vec{\mathbf{r}}] = \begin{pmatrix} y_1 \cdot z_2 - y_2 \cdot z_1 \\ x_2 \cdot z_1 - x_1 \cdot z_2 \\ x_1 \cdot y_2 - x_2 \cdot y_1 \end{pmatrix} = \\ = \begin{pmatrix} z_0(\gamma\sin(\theta) + (\gamma\theta + d/2)\cos(\theta)) \\ z_0((\gamma\theta + d/2)\sin(\theta) - \gamma\cos(\theta)) \\ (\gamma\theta + d/2)^2 - \gamma(x_0\sin(\theta) - y_0\cos(\theta)) - (\gamma\theta + d/2)(x_0\cos(\theta) + y_0\sin(\theta)) \end{pmatrix} \\ |\vec{\mathbf{r}}| = \sqrt{x_2^2 + y_2^2 + z_2^2} = \sqrt{x_0^2 + y_0^2 + z_0^2 + (\gamma\theta + d/2)^2 - 2(\gamma\theta + d/2)(x_0\cos(\theta) + y_0\sin(\theta))} \end{cases} \tag{4}$$

Submitting Eq. (4) into Eq. (3), we obtain the magnetic induction components:

$$\begin{cases} B_x(\vec{\mathbf{r}}_0) = \frac{\mu_0 I z_0}{4\pi}\int_{\theta_1}^{\theta_2} \frac{\gamma\sin(\theta)+(\gamma\theta+d/2)\cos(\theta)}{\left(x_0^2+y_0^2+z_0^2+(\gamma\theta+d/2)^2-2(\gamma\theta+d/2)(x_0\cos(\theta)+y_0\sin(\theta))\right)^{3/2}} d\theta \\ B_y(\vec{\mathbf{r}}_0) = \frac{\mu_0 I z_0}{4\pi}\int_{\theta_1}^{\theta_2} \frac{(\gamma\theta+d/2)\sin(\theta)-\gamma\cos(\theta)}{\left(x_0^2+y_0^2+z_0^2+(\gamma\theta+d/2)^2-2(\gamma\theta+d/2)(x_0\cos(\theta)+y_0\sin(\theta))\right)^{3/2}} d\theta \\ B_z(\vec{\mathbf{r}}_0) = \frac{\mu_0 I}{4\pi}\int_{\theta_1}^{\theta_2} \frac{(\gamma\theta+d/2)^2-\gamma(x_0\sin(\theta)-y_0\cos(\theta))-(\gamma\theta+d/2)(x_0\cos(\theta)+y_0\sin(\theta))}{\left(x_0^2+y_0^2+z_0^2+(\gamma\theta+d/2)^2-2(\gamma\theta+d/2)(x_0\cos(\theta)+y_0\sin(\theta))\right)^{3/2}} d\theta \end{cases} \tag{5}$$

Later, we will need only one of the field components, $B_z(\vec{\mathbf{r}}_0)$, which can be calculated analytically on the central axis perpendicular to the spiral plane ($x_0 \equiv 0, y_0 \equiv 0$):

$$B_z(z_0) = \frac{\mu_0 I}{4\pi}\int_{\theta_1}^{\theta_2} \frac{(\gamma\theta + d/2)^2 d\theta}{((\gamma\theta + d/2)^2 + z_0^2)^{3/2}} \Rightarrow \begin{Bmatrix} t = \gamma\theta + d/2 \\ dt = \gamma d\theta \end{Bmatrix} \Rightarrow \frac{\mu_0 I}{4\pi\gamma}\int_{\gamma\theta_1 + d/2}^{\gamma\theta_2 + d/2} \frac{t^2 dt}{(t^2 + z_0^2)^{3/2}} =$$

$$= \frac{\mu_0 I}{4\pi\gamma}\left(\ln\left(t + \sqrt{t^2 + z_0^2}\right) - \frac{t}{\sqrt{t^2 + z_0^2}}\right)\Bigg|_{\gamma\theta_1 + d/2}^{\gamma\theta_2 + d/2} = \tag{6}$$

$$= \frac{\mu_0 I}{4\pi\gamma}\left(\ln\left(\frac{2\gamma\theta_2 + d + \sqrt{(2\gamma\theta_2 + d)^2 + 4z_0^2}}{2\gamma\theta_1 + d + \sqrt{(2\gamma\theta_1 + d)^2 + 4z_0^2}}\right) + \frac{2\gamma\theta_1 + d}{\sqrt{(2\gamma\theta_1 + d)^2 + 4z_0^2}} - \frac{2\gamma\theta_2 + d}{\sqrt{(2\gamma\theta_2 + d)^2 + 4z_0^2}}\right)$$

In Eq. (6), all lengths are measured in meters, angles are in radians, and current is in amperes. The calculated magnetic induction will be in units of Tesla. To model a spiral coil wound turn-to-turn, we set $\gamma = d/(2\pi)$. We can also consider the gap between the metal cores of the wires, which encompasses the isolation material surrounding the cores. In this scenario, we have $\gamma = (d_0 + 2\delta)/(2\pi)$, where d_0 signifies the wire metal core, and δ denotes the half gap between the wire metal cores. In Eq. (6), d should be substituted with $d = d_0 + 2\delta$. In designs involving enamelled wires, it is usually acceptable to neglect the isolation thickness ($2\delta \ll d$).

3. Solenoid coil

Initially, our assumption was that the shimming coil, once optimized, would be comprised of a stack of short solenoids. However, it has become evident that such a design would not offer the required level of accuracy for compensating residual induction. Nevertheless, we have decided to retain this section in our report as it employs similar calculation methods and remains relevant within the context of our research. Fig. 4 shows the geometrical model of a single-layer solenoidal coil. The vector and scalar quantities required for calculating the magnetic induction in Eq. (2) can be expressed in terms of the radius vector pointing to the wire axis:

$$\begin{cases} \vec{\mathbf{r}}_{d\theta} = \begin{pmatrix} (R + d/2)\cos(\theta) \\ (R + d/2)\sin(\theta) \\ \gamma\theta \end{pmatrix} \\ \vec{\boldsymbol{\chi}} = \frac{d\vec{\mathbf{r}}_{d\theta}}{d\theta} \times \frac{1}{\left|\frac{d\vec{\mathbf{r}}_{d\theta}}{d\theta}\right|} = \frac{1}{\sqrt{(R+d/2)^2 + \gamma^2}} \times \begin{pmatrix} -(R + d/2)\sin(\theta) \\ (R + d/2)\cos(\theta) \\ \gamma \end{pmatrix} \\ dl = d\theta\left|\frac{d\vec{\mathbf{r}}_{d\theta}}{d\theta}\right| = d\theta\sqrt{(R + d/2)^2 + \gamma^2} \\ \vec{\mathbf{r}} = \vec{\mathbf{r}}_0 - \vec{\mathbf{r}}_{d\theta} = \begin{pmatrix} x_0 - (R + d/2)\cos(\theta) \\ y_0 - (R + d/2)\sin(\theta) \\ z_0 - \gamma\theta \end{pmatrix} \end{cases} \tag{7}$$

In Eq. (7), R denotes the internal diameter of the coil, d is the wire diameter, and $\gamma \geq d/(2\pi)$ is the interturn parameter that synchronizes the linear displacement along the z-axis with the rotation.

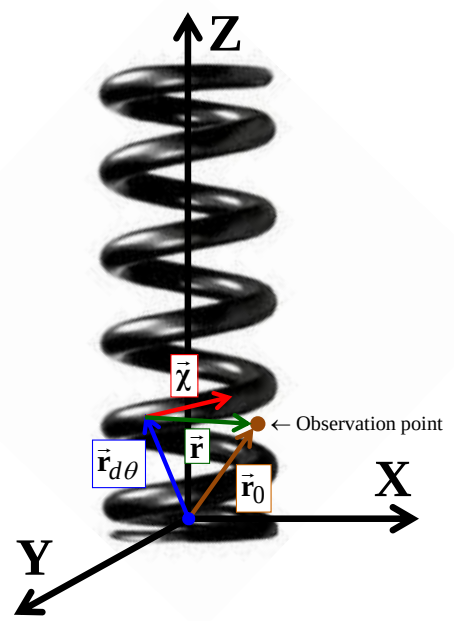


Fig. 4. Geometrical model of a single layer solenoidal coil.

For a single-layer solenoidal coil symmetrical along the z-axis with respect to the origin of the coordinate system, the magnetic induction can be calculated as follows:

$$\vec{\mathbf{B}}(\vec{\mathbf{r}}_0) = \frac{\mu_0}{4\pi} I \int_L \frac{[\vec{\chi}\times\vec{\mathbf{r}}]dl}{|\vec{\mathbf{r}}|^3} = \frac{\mu_0 I}{4\pi} \int_{-\psi}^{\psi} \frac{[\vec{\mathbf{v}}\times\vec{\mathbf{r}}]d\theta}{|\vec{\mathbf{r}}|^3} \tag{8}$$

where

$$\begin{cases} \vec{\mathbf{v}} = \begin{pmatrix} -(R+d/2)\sin(\theta) \\ (R+d/2)\cos(\theta) \\ \gamma \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \\ \vec{\mathbf{r}} = \begin{pmatrix} x_0 - (R+d/2)\cos(\theta) \\ y_0 - (R+d/2)\sin(\theta) \\ z_0 - \gamma\theta \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \\ [\vec{\mathbf{v}} \times \vec{\mathbf{r}}] = \begin{pmatrix} y_1 \cdot z_2 - y_2 \cdot z_1 \\ x_2 \cdot z_1 - x_1 \cdot z_2 \\ x_1 \cdot y_2 - x_2 \cdot y_1 \end{pmatrix} \\ [\vec{\mathbf{v}} \times \vec{\mathbf{r}}] = \begin{pmatrix} -\gamma y_0 + (R+d/2)(z_0\cos(\theta) - \gamma\theta\cos(\theta) + \gamma\sin(\theta)) \\ \gamma x_0 + (R+d/2)(z_0\sin(\theta) - \gamma\theta\sin(\theta) - \gamma\cos(\theta)) \\ (R+d/2)^2 - (R+d/2)(y_0\sin(\theta) + x_0\cos(\theta)) \end{pmatrix} \\ |\vec{\mathbf{r}}| = \sqrt{x_2^2 + y_2^2 + z_2^2} = \sqrt{x_0^2 + y_0^2 + (z_0 - \gamma\theta)^2 + (R+d/2)^2 - 2(R+d/2)(x_0\cos(\theta) + y_0\sin(\theta))} \end{cases} \tag{9}$$

Regarding the direction of positive contour traversal and angles, we will adhere to the conventions established in Fig. 3.

By utilizing Eqs. (8) and (9), we can derive the expression for the component $B_z(\vec{\mathbf{r}}_0)$:

$$B_z(\vec{\mathbf{r}}_0) = \frac{\mu_0 I}{4\pi}\int_{-\psi}^{\psi} \frac{\left((R+d/2)^2-(R+d/2)(y_0\sin(\theta)+x_0\cos(\theta))\right)d\theta}{\left(x_0^2+y_0^2+(z_0-\gamma\theta)^2+(R+d/2)^2-2(R+d/2)(x_0\cos(\theta)+y_0\sin(\theta))\right)^{3/2}} \tag{10}$$

Integral in Eq. (10) can be calculated analytically along the coil axis ($x_0 \equiv 0, y_0 \equiv 0$):

$$B_z(z_0) = \frac{\mu_0 I (R+d/2)^2}{4\pi}\int_{-\psi}^{\psi} \frac{d\theta}{((R+d/2)^2+(z_0-\gamma\theta)^2)^{3/2}} = -\frac{\mu_0 I}{4\pi\gamma}\left(\frac{z_0-\gamma\theta}{\sqrt{(R+d/2)^2+(z_0-\gamma\theta)^2}}\right)\Bigg|_{-\psi}^{\psi} =$$

$$= \frac{\mu_0 I}{4\pi\gamma}\left(\frac{z_0+\gamma\psi}{\sqrt{(R+d/2)^2+(z_0+\gamma\psi)^2}} - \frac{z_0-\gamma\psi}{\sqrt{(R+d/2)^2+(z_0-\gamma\psi)^2}}\right) \tag{11}$$

$\psi = \pi N$, where N is the number of turns in a layer.

All the lengths are measured in the meters, the angles are in the radians, and the current is in the amperes. The calculated magnetic induction will be in units of Tesla. The coordinate z_0 can be both inside and outside the coil. For a solenoidal coil wound turn-to-turn, we must put $\gamma = d/(2\pi)$. We can also consider the gap between the metal cores of the wires, which encompasses the isolation material surrounding the cores. In this scenario, we have $\gamma = (d_0 + 2\delta)/(2\pi)$, where d_0 signifies the wire metal core, and δ denotes the half gap between the wire metal cores. In Eq. (11), d should be substituted with $d = d_0 + 2\delta$. In designs involving enamelled wires, it is usually acceptable to neglect the isolation thickness ($2\delta \ll d$). The geometrical interpretation of Eq. (11) is shown in Fig. 5, where z_0 can be positive or negative relative to the midpoint of the coil.

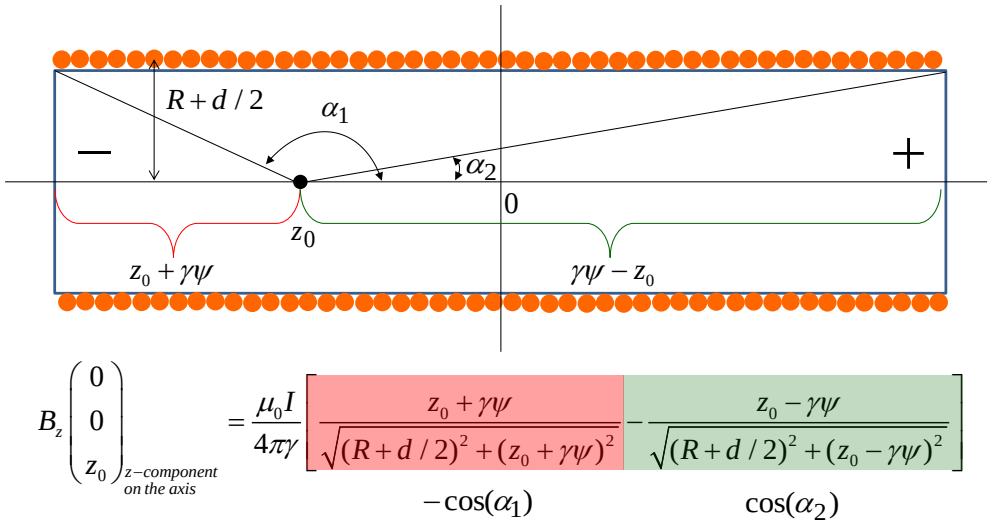


Fig. 5. Geometrical interpretation of Eq. (11).

For a solenoidal coil with the number of layers $M \geq 1$, we obtain from Eq. (11):

$$B_z(z_0) = \frac{\mu_0 I}{4\pi\gamma}\sum_{m=1}^{M}\left(\frac{z_0+\gamma\psi}{\sqrt{(R+d(2m-1)/2)^2+(z_0+\gamma\psi)^2}} - \frac{z_0-\gamma\psi}{\sqrt{(R+d(2m-1)/2)^2+(z_0-\gamma\psi)^2}}\right) \quad (12)$$

We have developed a Python algorithm that calculates the magnetic field strength $H_z = B_z/\mu_0$ as a function of z_0 using Eq. (12), which is saved in the Appendix. Fig. 6 shows an example of field distribution inside a solenoid with the following parameters:

```
The solenoid is symmetrical about 0: z0 = [-maxz0, maxz0]
Enter the solenoid length L in meters: 0.2
Enter the solenoid internal radius R in meters: 0.01
Enter the wire diameter d0 without isolation in meters: 0.001
Enter the wire isolation thickness in meters: 0.0
Enter the number M of winding layers: 4
Enter the range maxz0 for z0 = [-maxz0, maxz0] in meters: 0.1
Enter the current I in amperes: 1
```

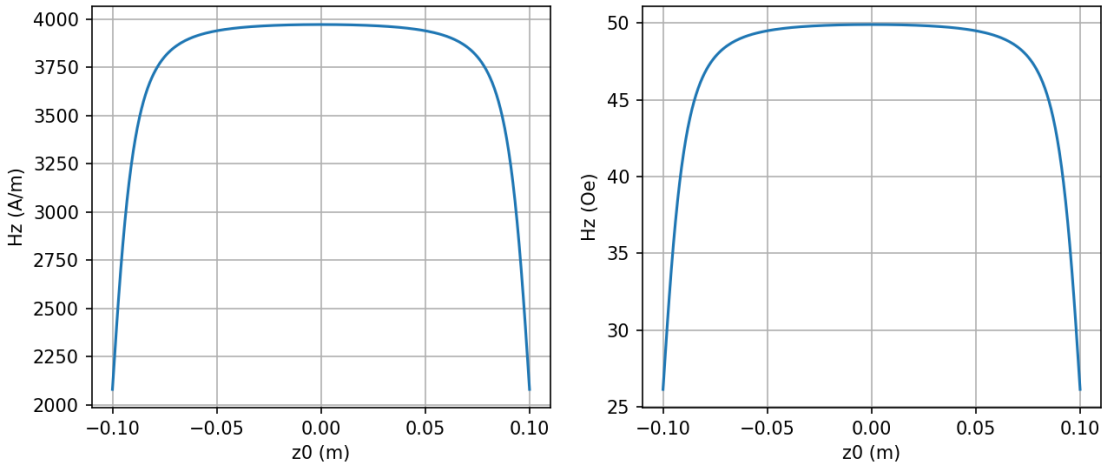


Fig. 6. An example of field distribution inside a solenoid: A/m for SI and Oe for cgs.

4. Shimming coil design and optimisation

To compensate for a given magnetic induction profile along an axis, a stack of short solenoidal coils can be used. Designing such a system requires an optimisation algorithm. We have developed an optimisation algorithm that has no restrictions on the number of coils and provides a theoretical limit of accuracy for the 1D design. Our approach involves considering adjacent layers of planar spirals, which are oriented perpendicular to the axis of the magnetic induction. Each spiral is characterized by two angles $(\theta_{1m}, \theta_{2m})$, its individual feeding current I_m, and a numeration index m. The angles θ_{1m} determine the size of the internal hole in the spiral, which corresponds to the future coil's inner diameter. Therefore, the parameters θ_{1m} will remain constant and be equal in absolute value. The optimisation vector for this study will consist of the parameters (θ_{2m}, I_m). It should be

noted that during optimisation, some of the values of θ_{2m} may become negative, such as in the case of an anticlockwise spiral. In such scenarios, the corresponding θ_{1m} must also be made negative while preserving its absolute value.

As the number of turns in the coil and the current magnitude have an equivalent effect on the field magnitude, it is recommended to conduct two types of optimisations. In the first type, shown in Fig. 7, the number of turns in each spiral is held constant, such as having an identical value, and only the current values are optimized. In the second type, shown in Fig. 8, the current values are fixed, and the number of turns in each coil is optimized. In order to implement a design with an optimized current profile, a more complex current driver is required to power the coils. On the other hand, optimizing the number of turns while keeping the same current simplifies the current driver. However, the manufacturing process for the coils becomes more complex since they now have varying numbers of turns.

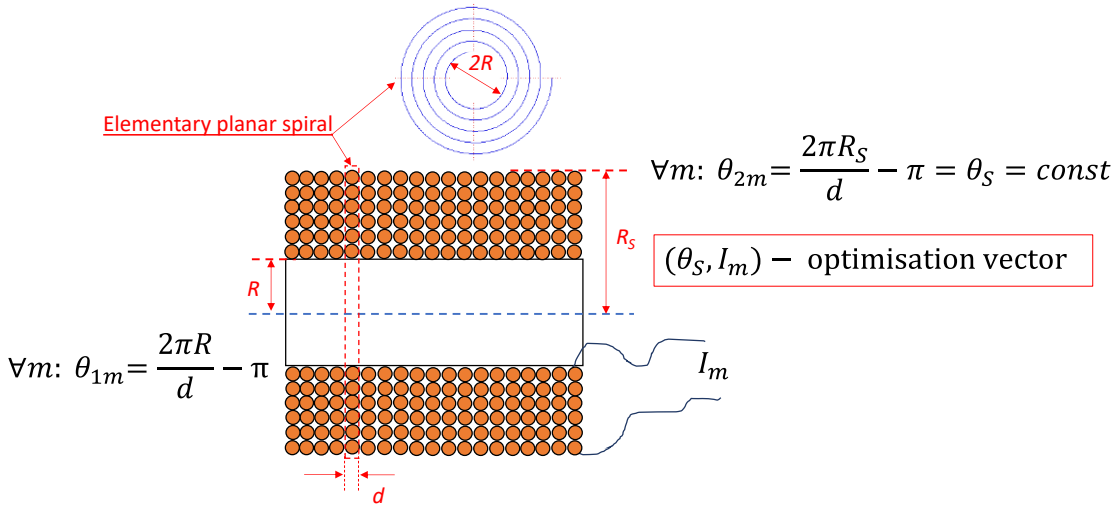


Fig. 7. The first optimisation scheme, where the number of turns is fixed, and only the currents are optimized.

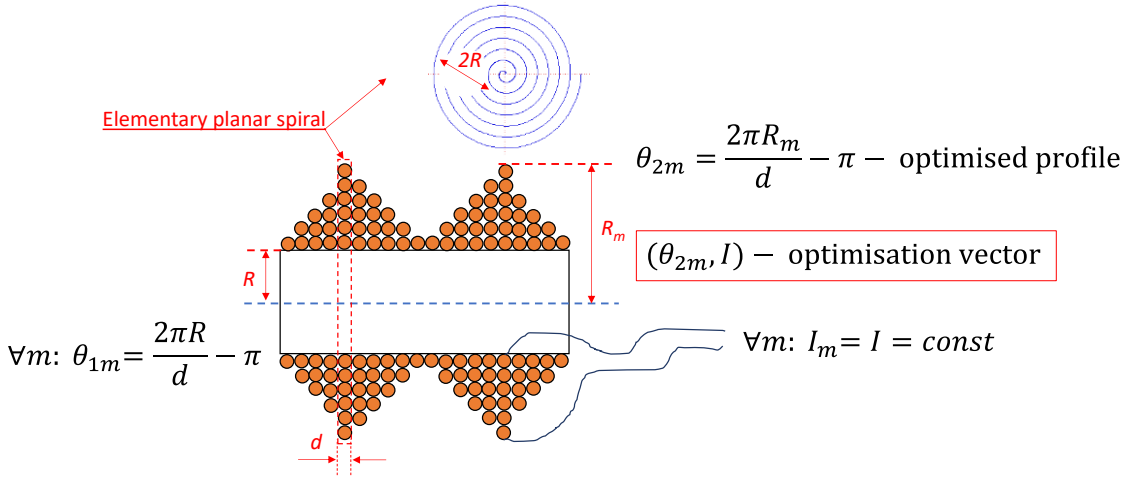


Fig. 8. The second optimisation scheme, where the current values are fixed, and only the turns are optimized.

Once the optimal profile of (θ_S, I_m) or (θ_{2m}, I) has been achieved, the next step in practical implementation is to combine the elementary spiral coils into a stack.

In Eq. (6), the induction was calculated from a single spiral coil placed in the beginning of coordinate system. For a set of spiral coils placed at z_m, the total induction B_z in an observation point z_0 can be expressed through the following sum:

$$B_z\left(z_0, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\theta}}_2, \vec{\mathbf{I}}\right) = \frac{\mu_0}{4\pi\gamma}\sum_{m=1}^{M} I_m \left(\begin{array}{l} \ln\left(\frac{2\gamma\theta_{2m}+d+\sqrt{(2\gamma\theta_{2m}+d)^2+4(z_0-z_m)^2}}{2\gamma\theta_{1m}+d+\sqrt{(2\gamma\theta_{1m}+d)^2+4(z_0-z_m)^2}}\right) + \\ + \frac{2\gamma\theta_{1m}+d}{\sqrt{(2\gamma\theta_{1m}+d)^2+4(z_0-z_m)^2}} - \frac{2\gamma\theta_{2m}+d}{\sqrt{(2\gamma\theta_{2m}+d)^2+4(z_0-z_m)^2}} \end{array} \right) \quad (13)$$

where $\vec{\boldsymbol{\theta}}_1 = (\theta_{11}, \theta_{12}, \ldots, \theta_{1M})$ with $\theta_{1m} = 2\pi R/d - \pi = const$, $\vec{\boldsymbol{\theta}}_2 = (\theta_{21}, \theta_{22}, \ldots, \theta_{2M})$, and $\vec{\mathbf{I}} = (I_1, I_2, \ldots., I_M)$. We choose $M = \mathrm{int}(L/w)$ as the odd integer number, where L is the whole optimisation length and $w \geq d$ is the width of the spiral substrate, including the spiral winding. The induction in Eq. (13) is intended to compensate the residual axial field $B_0(z_0)$, where $z_0 \in [-L_0/2, L_0/2]$ and $L_0 \leq L$. For the coordinates of the spiral coils, we choose $z_m = -(M-1)w/2 + \mathrm{w} \times (m-1),\ m = \overline{1, M}$.

For testing the algorithm, we use an experimental residual induction (Tesla) interpolated within $z_0 \in [-0.030, 0.030]$ m:

$$\begin{aligned} B_0(z_0) = {} & -5.859 \times 10^{-6} + 4.766114 \times 10^{-3} \times z_0 - 4.86506371 \times 10 \times z_0^2 - 1.4609783504 \times 10 \times z_0^3 + \\ & + 4.2600403748 \times 10^2 \times z_0^4 + 3.04437 \times 10^4 \times z_0^5 - 8.75637 \times 10^5 \times z_0^6 - 7.65903 \times 10^6 \times z_0^7 + \\ & + 1.46997 \times 10^8 \times z_0^8 \end{aligned} \quad (14)$$

In order to determine the optimal profile (θ_{2m}, I) or (θ_S, I_m) (refer to the optimisation schemes depicted in Figs. 7 and 8), which can deliver the minimum of the function $\left|B_0(z_0) + B_z\left(z_0, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\theta}}_2, \vec{\mathbf{I}}\right)\right|$ for any value of z_0, we will employ the non-linear least squares method to minimize the following multi-dimensional function:

$$J\left(\vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\theta}}_2, \vec{\mathbf{I}}\right) = \sum_{q=1}^{Q}\left(B_0(z_q) + B_z\left(z_q, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\theta}}_2, \vec{\mathbf{I}}\right)\right)^2 \quad (15)$$

where the summation is carried out over a set of $z_0 = z_q$ uniformly distributed within $[-L_0/2, L_0/2]$. The sampling step for this set can be selected to be equal to the width w of the spiral substrate. Choosing a smaller value would not be meaningful, as the spatial resolution in the problem is determined by w. Therefore, we refrain from assigning an additional index (z_{0q}) to this set of points. The positive z-axis will align with the positive direction of the induction generated by the positive current flow. It is imperative to carefully consider the orientation of the spirals in relation to the residual field, taking into account their winding (see Fig. 3).

We will conduct two distinct optimisations: one for the current $\vec{\mathbf{I}}$ and another for the number of turns $\vec{\boldsymbol{\theta}}_2$ (see Figs. 7 and 8). Thus, the functional in Eq. (15) will be expressed using a single vector of independent variables denoted as $\vec{\boldsymbol{\beta}}$:

$$J(\vec{\boldsymbol{\beta}}) = \sum_{q=1}^{Q} \left(B_z(z_q, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\theta}}_2, \vec{\boldsymbol{\beta}}) + B_0(z_q) \right)^2 \tag{16}$$

or

$$J(\vec{\boldsymbol{\beta}}) = \sum_{q=1}^{Q} \left(B_z(z_q, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\beta}}, \vec{\mathbf{I}}) + B_0(z_q) \right)^2 \tag{17}$$

4.1 Unconstrained Newton minimisation method

To find the optimal vector $\vec{\boldsymbol{\beta}}$ that provides the minimum of the functionals in Eqs. (16) and (17) using the Newton method, the following iteration sequence $\vec{\boldsymbol{\beta}}^k \to \vec{\boldsymbol{\beta}}$ is used:

$$\vec{\boldsymbol{\beta}}^{k+1} = \vec{\boldsymbol{\beta}}^k - [\vec{\nabla}^2 J(\vec{\boldsymbol{\beta}}^k)]^{-1} \times \vec{\nabla} J(\vec{\boldsymbol{\beta}}^k), \quad k \geq 0 \tag{18}$$

where

$$\vec{\nabla} J(\vec{\boldsymbol{\beta}}^k) = \left\{ \frac{\partial J(\vec{\boldsymbol{\beta}}^k)}{\partial \beta_m^k} \right\}_{m=\overline{1,M}}^{\mathrm{T}} \tag{19}$$

represents the gradient calculated for $\vec{\boldsymbol{\beta}}^k = \{\beta_m^k\}_{m=\overline{1,M}}$ (the operator "T" denotes the transpose operation, which converts a row array into a column array),

$$[\vec{\nabla}^2 J(\vec{\boldsymbol{\beta}}^k)]^{-1} = \left\{ \frac{\partial^2 J(\vec{\boldsymbol{\beta}}^k)}{\partial \beta_m^k \partial \beta_n^k} \right\}_{m,n=\overline{1,M}}^{-1} \tag{20}$$

represents the inverse of the Hessian matrix, which contains the second partial derivatives of the objective function, and $\vec{\boldsymbol{\beta}}^0$ is an initial vector.

For the numerical implementation of the iteration process in Eq. (18), the vector $\vec{\mathbf{h}} = -[\vec{\nabla}^2 J(\vec{\boldsymbol{\beta}}^k)]^{-1} \times \vec{\nabla} J(\vec{\boldsymbol{\beta}}^k)$ should be searched at each iteration step as a solution to the linear equation $\vec{\nabla}^2 J(\vec{\boldsymbol{\beta}}^k) \times \vec{\mathbf{h}} = -\vec{\nabla} J(\vec{\boldsymbol{\beta}}^k)$ using a non-direct method. However, solving such a system for vectors of large dimensions can be computationally expensive and require a significant amount of memory, while convergence is not necessarily guaranteed. In such cases, it is necessary to use modified methods:

- *Truncated-Newton methods are a family of methods suitable for solving large nonlinear unconstrained optimisation problems. At each iteration k, the current estimate of the solution is updated (i.e., a step is computed) by approximately solving* ***the Newton equation*** $\vec{\nabla}^2 J(\vec{\boldsymbol{\beta}}^k) \times \vec{\mathbf{h}} = -\vec{\nabla} J(\vec{\boldsymbol{\beta}}^k)$ *using an iterative algorithm. This results in a doubly iterative method: an outer iteration* $\vec{\boldsymbol{\beta}}^{k+1} = \vec{\boldsymbol{\beta}}^k + \vec{\mathbf{h}}$ *for*

the nonlinear optimisation problem, and an inner iteration for the Newton linear equation. The inner iteration is typically stopped or "truncated" before the solution to the Newton equation is obtained.

- *For large-scale problems, it is sensible to use an iterative method to approximately solve the Newton equation. Moreover, it should be an iterative method with a variable tolerance, so that far away from the solution, the Newton equation is not solved to undue accuracy. Only when the solution is approached should we consider expending enough effort to compute something like the exact Newton direction. As we approach the solution, the Hessian $\vec{\nabla}^2 J(\vec{\beta}^k)$ will converge to $\vec{\nabla}^2 J(\vec{\beta})$, where $\vec{\beta}$ is the vector. Consequently, by exploiting information from previous iterations, it is possible that a closer approximation to the exact solution can be determined with no increase in effort.*

For numerical minimisation, we used the Python library scipy.optimize.minimize:

```
from scipy.optimize import minimize

result = minimize(fun, x0, method = 'Newton-CG', jac = GJ, hess = HJ, tol = tolerance)
output = np.array(result.x)
```

Here, fun is Eq. (16) or (17), x0 is the initial vector $\vec{\beta}^0$, jac $=$ GJ is the callable gradient function in Eq. (19) with an array argument, hess $=$ HJ is the callable Hessian function in Eq. (20) with an array argument (provided without its inversion), and tol $=$ tolerance is a small parameter ($10^{-6} \div 10^{-8}$) used to control the calculation accuracy during the iterative process. We have conducted testing on the "Newton-CG" method, which exhibited excellent performance in our experiments (references to the papers shown below can be found in the library description):

Method Newton-CG uses a Newton-CG algorithm [5] pp. 168 (also known as the truncated Newton method). It uses a CG method to the compute the search direction. See also TNC method for a box-constrained minimization with a similar algorithm. Suitable for large-scale problem.

Alternatively, you may explore other methods that require the provision of both the gradient and Hessian:

- *Method trust-ncg uses the Newton conjugate gradient trust-region algorithm [5] for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector. Suitable for large-scale problems.*
- *Method trust-krylov uses the Newton GLTR trust-region algorithm [14], [15] for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector. Suitable for large-scale problems. On indefinite problems it requires usually less iterations than the trust-ncg method and is recommended for medium and large-scale problems.*

- *Method trust-exact is a trust-region method for unconstrained minimization in which quadratic subproblems are solved almost exactly [13]. This algorithm requires the gradient and the Hessian (which is not required to be positive definite). It is, in many situations, the Newton method to converge in fewer iterations and the most recommended for small and medium-size problems.*

4.2 Shimming coil optimisation for the current driver

The number of turns in all spiral coils is fixed and the same (see Fig. 7). To obtain the components of the gradient of the functional in Eq. (16), we utilise Eq. (13):

$$\begin{cases} \frac{\partial J(\vec{\mathbf{I}})}{\partial I_m} = 2\sum_{q=1}^{Q}\left(B_z\left(z_q,\vec{\boldsymbol{\theta}}_1,\vec{\boldsymbol{\theta}}_2,\vec{\mathbf{I}}\right)+B_0\left(z_q\right)\right)\frac{\partial B_z\left(z_q,\vec{\boldsymbol{\theta}}_1,\vec{\boldsymbol{\theta}}_2\right)}{\partial I_m} \\ B_z\left(z_q,\vec{\boldsymbol{\theta}}_1,\vec{\boldsymbol{\theta}}_2,\vec{\mathbf{I}}\right) = \frac{\mu_0}{4\pi\gamma}\sum_{m=1}^{M} I_m\left(\ln\left(\frac{2\gamma\theta_{2m}+d+\sqrt{(2\gamma\theta_{2m}+d)^2+4(z_q-z_m)^2}}{2\gamma\theta_{1m}+d+\sqrt{(2\gamma\theta_{1m}+d)^2+4(z_q-z_m)^2}}\right)+\right. \\ \left.+\frac{2\gamma\theta_{1m}+d}{\sqrt{(2\gamma\theta_{1m}+d)^2+4(z_q-z_m)^2}}-\frac{2\gamma\theta_{2m}+d}{\sqrt{(2\gamma\theta_{2m}+d)^2+4(z_q-z_m)^2}}\right) \\ \frac{\partial B_z\left(z_q,\vec{\boldsymbol{\theta}}_1,\vec{\boldsymbol{\theta}}_2\right)}{\partial I_m} = \frac{\mu_0}{4\pi\gamma}\left(\ln\left(\frac{2\gamma\theta_{2m}+d+\sqrt{(2\gamma\theta_{2m}+d)^2+4(z_q-z_m)^2}}{2\gamma\theta_{1m}+d+\sqrt{(2\gamma\theta_{1m}+d)^2+4(z_q-z_m)^2}}\right)+\right. \\ \left.+\frac{2\gamma\theta_{1m}+d}{\sqrt{(2\gamma\theta_{1m}+d)^2+4(z_q-z_m)^2}}-\frac{2\gamma\theta_{2m}+d}{\sqrt{(2\gamma\theta_{2m}+d)^2+4(z_q-z_m)^2}}\right) \end{cases} \tag{21}$$

For the Hessian's components, we obtain:

$$\frac{\partial^2 J(\vec{\mathbf{I}})}{\partial I_m \partial I_n} = 2\sum_{q=1}^{Q}\frac{\partial B_z\left(z_q,\vec{\boldsymbol{\theta}}_1,\vec{\boldsymbol{\theta}}_2\right)}{\partial I_n}\times\frac{\partial B_z\left(z_q,\vec{\boldsymbol{\theta}}_1,\vec{\boldsymbol{\theta}}_2\right)}{\partial I_m} \tag{22}$$

So, the symmetric matrix remains unchanged regardless of the iterative variable $\vec{\boldsymbol{\beta}}^k$. Given the values of $\theta_{1m} = 2\pi R/d - \pi$ and $\theta_{2m} = 2\pi R_S/d - \pi$, where R and R_S are the fixed radii for all spirals (see Fig. 7), we will assign the same initial value I^0 to all I_m^0 to start iterations in Eq. (18) based on the following criterion:

$$\overline{B_0\left(z_q\right)} = -B_z\left(0,\vec{\boldsymbol{\theta}}_1,\vec{\boldsymbol{\theta}}_2,\vec{\mathbf{I^0}}\right) \rightarrow I^0 = \frac{-4\pi\gamma\overline{B_0\left(z_q\right)}}{\mu_0\sum_{m=1}^{M}\left(\ln\left(\frac{2\gamma\theta_{2m}+d+\sqrt{(2\gamma\theta_{2m}+d)^2+4z_m^2}}{2\gamma\theta_{1m}+d+\sqrt{(2\gamma\theta_{1m}+d)^2+4z_m^2}}\right)+\right.\left.+\frac{2\gamma\theta_{1m}+d}{\sqrt{(2\gamma\theta_{1m}+d)^2+4z_m^2}}-\frac{2\gamma\theta_{2m}+d}{\sqrt{(2\gamma\theta_{2m}+d)^2+4z_m^2}}\right)} \tag{23}$$

where the overline denotes the average of the measured induction across all sampling points.

Numerical experiments demonstrate that optimisation under different initial conditions yields a wide range of acceptable current profiles. Hence, it is important to consider I^0 solely as a preliminary factor. By incorporating a broader range of criteria, such as the maximum driver current and coil power dissipation, it becomes possible to manipulate this parameter in order to achieve varying current profiles.

4.2.1 PWM driver without current amplification

Using a stack of a few short solenoids will not yield the desired level of accuracy for compensating residual induction. To achieve global optimisation across the entire segment where induction was measured, a substantial number of spiral coils with individual current values is required. Utilizing digital-to-analog converters (DACs) would incur excessive costs, rendering this approach financially impractical. In this case, the preferred approach is to utilize a programmable pulse width modulation (PWM) multichannel driver along with averaging RC filters to generate DC outputs with minimal ripple. The permissible current range for a driver employing operational amplifiers (op amps) will be confined to $25 - 30$ mA per channel.

The fabrication of planar spiral coils, regardless of whether a substrate is used ($w > d$) or not ($w = d$), should not pose any significant challenges. These technologies have already been well-established for charging coils, as demonstrated in Fig. 9. A wire planar coil provides the advantage of achieving a tighter winding with a greater number of turns in comparison to a microstrip coil implemented on either a rigid or flexible PCB. Additionally, it may present a cost-effective option for manufacturing. If the wire diameter is not too small, it is possible to bond the turns together, enabling the coil to support itself. By employing this method, it is possible to attain the utmost packing density, thereby establishing a theoretical accuracy limit (for a given wire diameter) to compensate for residual induction.

Fig. 9. These exemplars of planar charging coils offer valuable insights, as their manufacturing technologies hold significant promise for fulfilling our objectives.

The selection of coil parameters will consider the restrictions imposed on the sample and measurement conditions, as well as the characteristics of the current driver. In this context, it would be useful to estimate the resistance of an individual spiral, which requires determining its length:

$$l = \int_{\theta_1}^{\theta_2} \sqrt{\gamma^2 + (\gamma\theta + d/2)^2}\, d\theta = (1/\gamma) \int_{\gamma\theta_1 + d/2}^{\gamma\theta_2 + d/2} \sqrt{\gamma^2 + x^2} dx = \left\{ \int \sqrt{\gamma^2 + x^2} dx = \frac{x\sqrt{\gamma^2 + x^2}}{2} + \frac{\gamma^2}{2} \ln\left(x + \sqrt{\gamma^2 + x^2}\right) \right\} = \frac{(2\gamma\theta_2 + d)\sqrt{4\gamma^2 + (2\gamma\theta_2 + d)^2}}{8\gamma} - \frac{(2\gamma\theta_1 + d)\sqrt{4\gamma^2 + (2\gamma\theta_1 + d)^2}}{8\gamma} + \frac{\gamma}{2} \ln\left(\frac{2\gamma\theta_2 + d + \sqrt{4\gamma^2 + (2\gamma\theta_2 + d)^2}}{2\gamma\theta_1 + d + \sqrt{4\gamma^2 + (2\gamma\theta_1 + d)^2}} \right) \quad (24)$$

Here, $d = d_0 + 2\delta$, where d_0 stands for the wire diameter without isolation or microstrip width, and δ represents the half gap between the turns, encompassing the isolation in the case of the wire coil. The resistances of the wire and microstrip spirals are determined as follows: for the wire, $r = \rho \frac{4l}{\pi d_0^2}$, and for the microstrip, $r = \rho \frac{l}{h d_0}$, where ρ (Ω·m) represents the resistivity of the conductor ($\sim 1.7 \times 10^{-8}$ Ω·m for copper). Additionally, the determination of the inductance of the spiral may be also required for modelling the current driver.

The process of current optimisation is implemented in the NewtonCG.py module. In our analysis, we will specifically consider a scenario where the substrate thickness w, inclusive of the turns, amounts to 0.5 mm, and the enamelled wire has a diameter of 101 microns (AWG 38). The coil's inner diameter measured 30 mm, while the outer diameter was 42 mm. Consequently, the parameters listed below were generated for each spiral coil, with the initial current value calculated using Eq. (23):

internal radius $R = 15$ mm

external radius $R_S = 21$ mm

number of turns 59

wire length $l = 6.67$ m

wire resistance $r = 14.14\ \Omega$

coil inductance $L = 211.58$ μH

Fig. 10 demonstrates the residual induction $B_0(z_0)$ interpolated by Eq. (14) (blue curve) and the simulated compensation result $B_0(z_0) + B_z\left(z_0, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\theta}}_2, \vec{\mathbf{I}}\right)$ (green curve). The stack of spiral coils occupied a length of 100 mm, exceeding the measured length for induction of 60 mm. This extended compensation length is crucial for achieving global optimisation. Thus, considering the coil thickness of 0.5 mm, the total of 201 coils will be required to fully occupy the compensation length of 100 mm. Fig. 11 illustrates the current profile required to achieve the optimisation presented in Fig. 10. The profile exhibits a substantial level of complexity; however, it effectively remains within a manageable range below 30 mA. Consequently, this characteristic enables its implementation using standard low-power operational amplifier stages without the necessity of additional current

boosters. Given that the shimming coil is intended for operation at cryogenic temperatures, the power dissipation of the coil emerges as an important design parameter. In the case of the discussed design, the power dissipation measured at 0.26 W.

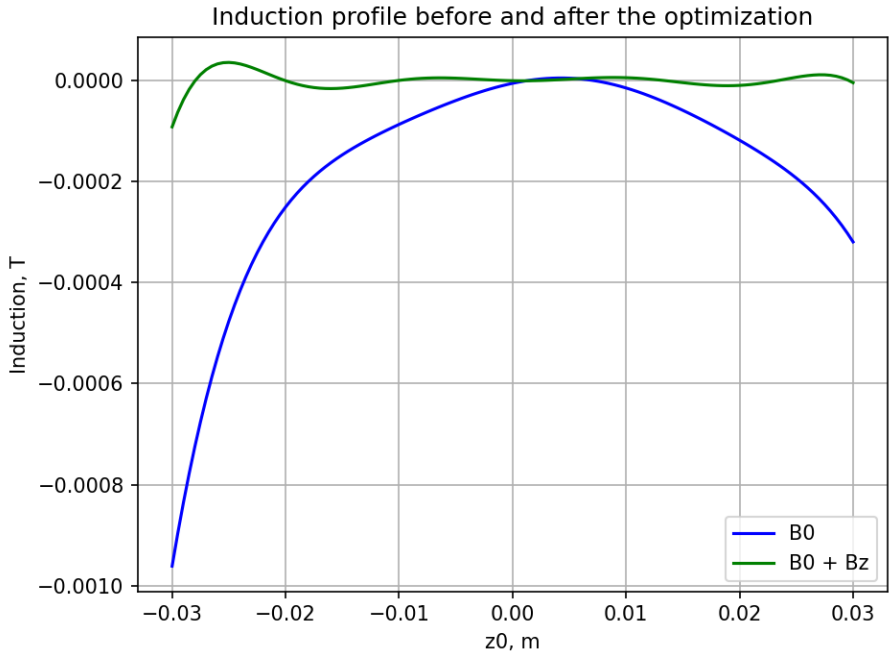


Fig. 10. Comparison of the residual induction profile before and after optimisation.

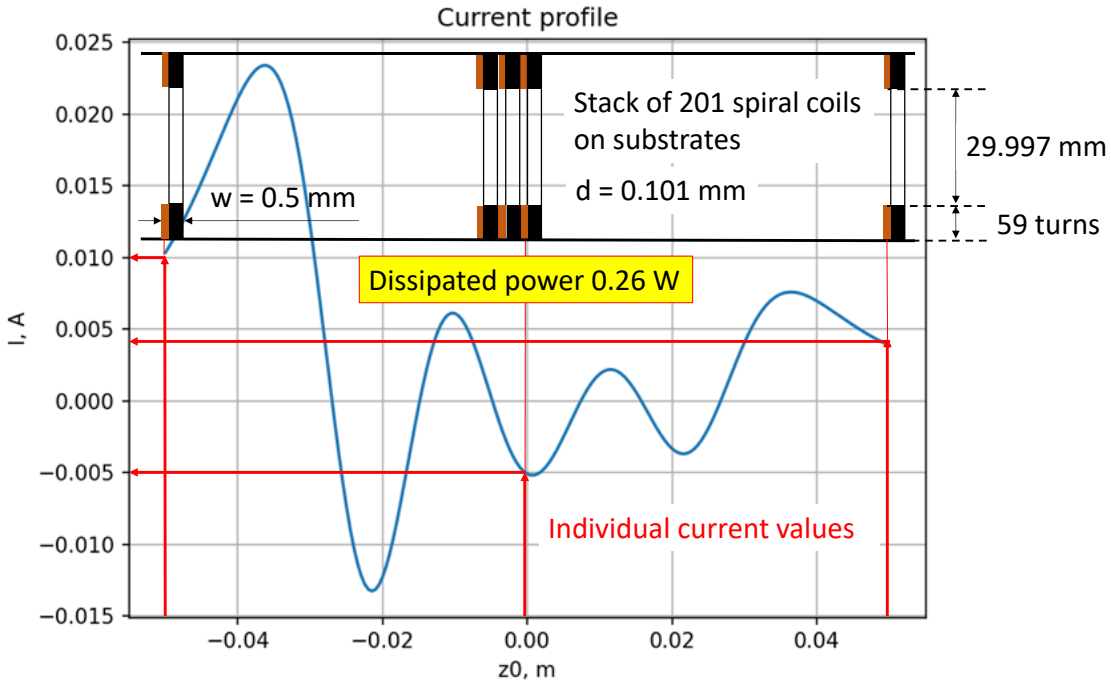
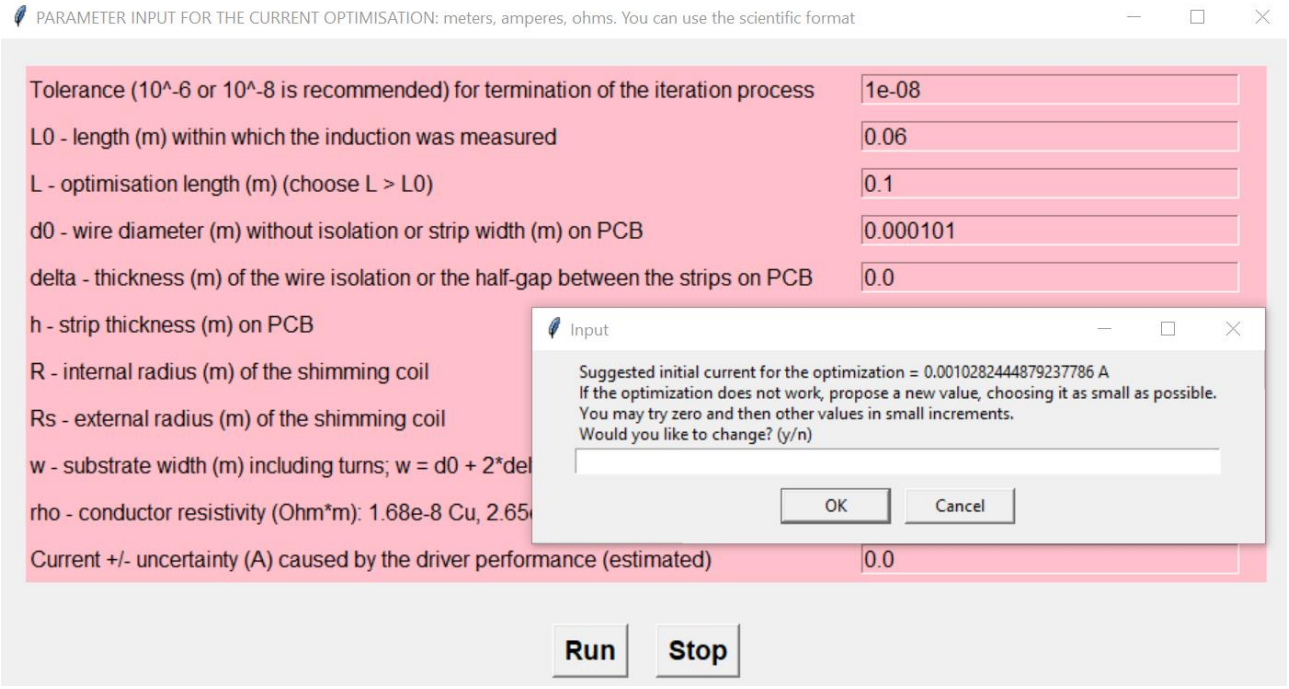
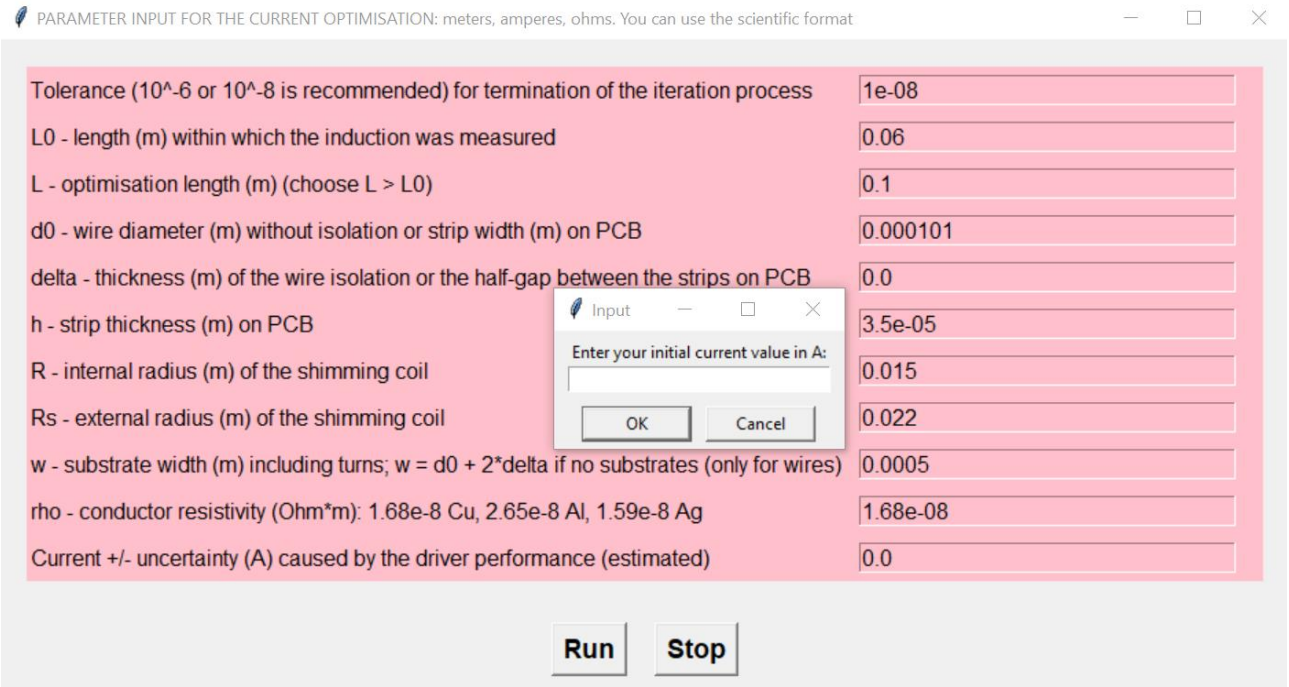


Fig. 11. The current profile necessary for achieving the optimisation in Fig. 10.

The outcome of the optimisation process is highly influenced by the initial values of the currents (see Eq. (23)). To facilitate this, the program prompts two dialog boxes for user input:



It is suggested to commence the optimisation process using the current value calculated according to Eq. (23) as the initial value, while allowing users the flexibility to propose alternative values. Experimenting with different initial data is crucial for obtaining a diverse range of solutions.



Therefore, while there is always a possibility of finding an optimal solution, it is essential to recognize that alternative solutions exist. When experimenting with the initial current values, careful attention should be given to the resulting current amplitudes and power dissipation. For instance, when considering the following parameters for the coil:

wire diameter $d = 0.101$ mm

internal radius $R = 15$ mm

external radius $R_S = 22$ mm

number of turns 69

wire length $l = 8$ m

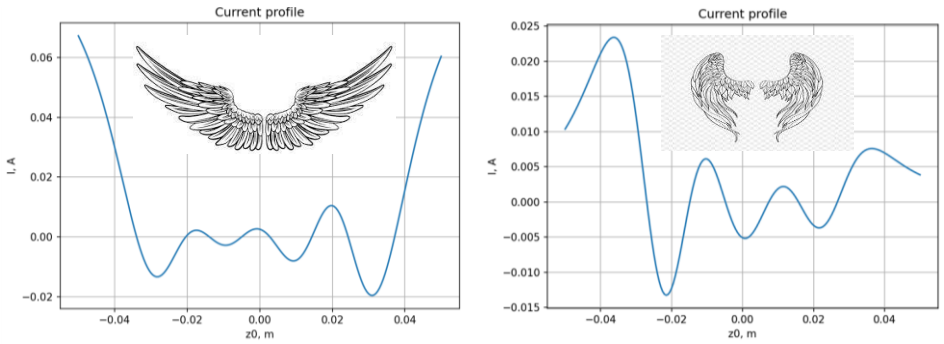
wire resistance $r = 17\ \Omega$

substrate thickness $w = 0.5$ mm (including the wire diameter)

number of the spiral coils 201

and utilizing the initial approximation $I^0 = 0.1$ A, it becomes possible to achieve excellent optimisation results, as demonstrated in Fig. 12, by employing a simpler current profile. However, it is important to note that this approach leads to higher current amplitudes and increased power dissipation up to 1.8 W. Consequently, implementing such a driver solely with ordinary operational amplifiers is no longer viable, necessitating the incorporation of additional current amplifier stages. The subsequent sections of this paper will delve into a detailed discussion of this particular design.

A noteworthy observation can be made when comparing the two solutions depicted in Figs. 11 and 12. The solution illustrated in Fig. 11 can be interpreted as folding the monotonic left- and right-wing sections observed in Fig. 12. Consequently, compensation for the residual induction provided can be achieved through either primarily utilizing monotonic sections of the current profile with heightened amplitude, or by employing folded current profiles. This observation offers a valuable insight into the potential approaches for achieving the desired compensation. It is truly remarkable that the algorithm successfully revealed these two scenarios.



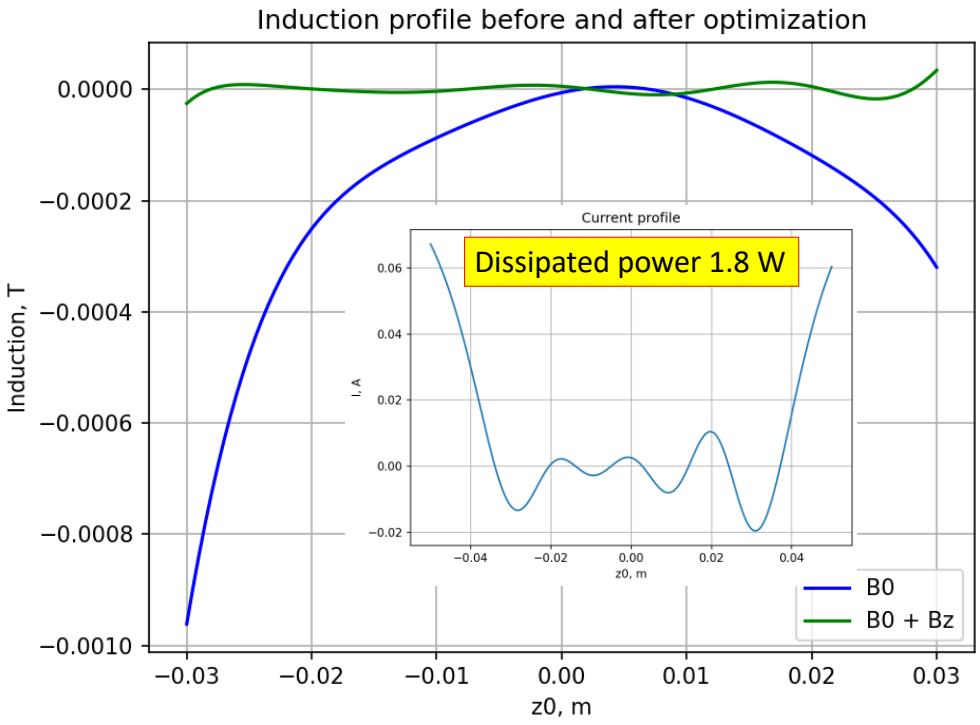


Fig. 12. Optimisation example with a varied initial current value. Excellent outcome achieved with a simplified current profile, albeit accompanied by significantly higher current amplitudes.

The idea of supplying individual currents to numerous coils may initially seem daunting. However, the solution we propose demonstrates that this is not a problem at all. The electrical current driver, illustrated in Fig. 13, makes use of PWM sources and RC filters while exhibiting programmable and self-contained attributes. Each PWM source generates a continuous chain of pulses with a duration of τ and a period of T. The zero harmonic (DC level) is determined by the ratio $(\tau/T) \times 5\text{V}$, where $(\tau/T)[\%]$ is called the duty cycle and 5V represents the amplitude of the pulses. To eliminate non-zero harmonics in the PWM spectrum, averaging is employed, achieved through an electrolytic capacitor $C_0 = 100$ μF. Tantalum or niobium electrolytic capacitors would offer the most compact form factor, while delivering superior performance and longer lifespan. The effectiveness of averaging depends on both the internal resistance of the PWM source and the load resistance, with higher values leading to improved results. We have incorporated a series resistance of $220\ \Omega$ at the output of the PWM source, which will also be present in the multi-channel PWM driver discussed later. Then, the voltage output of the excitation stage is connected to the input of a non-inverting voltage follower (buffer), implemented using an operational amplifier (op-amp) with short-connected negative feedback. A buffer features a significant input resistance (in MΩ range), unity gain, and a diminutive output resistance. This configuration effectively isolates the excitation stage from the load electrically. The maximum constant current of a typical low-power op-amp does not exceed 30 mA,

necessitating the use of a current-limiting resistor $R_1 = 170\ \Omega$ to maintain it at 25 mA. In a practical circuit, it is imperative to take into account the resistance of the wires connecting the driver to the coils. To prevent the op-amp from entering the saturation regime, it is advisable to utilize a DC power supply with a slightly higher voltage, such as 6 V. The direction of the current flow will be dictated by how the coil's ends are connected to the positive terminal and ground. For guidance on establishing the correct connection sequence, consult Fig. 3.

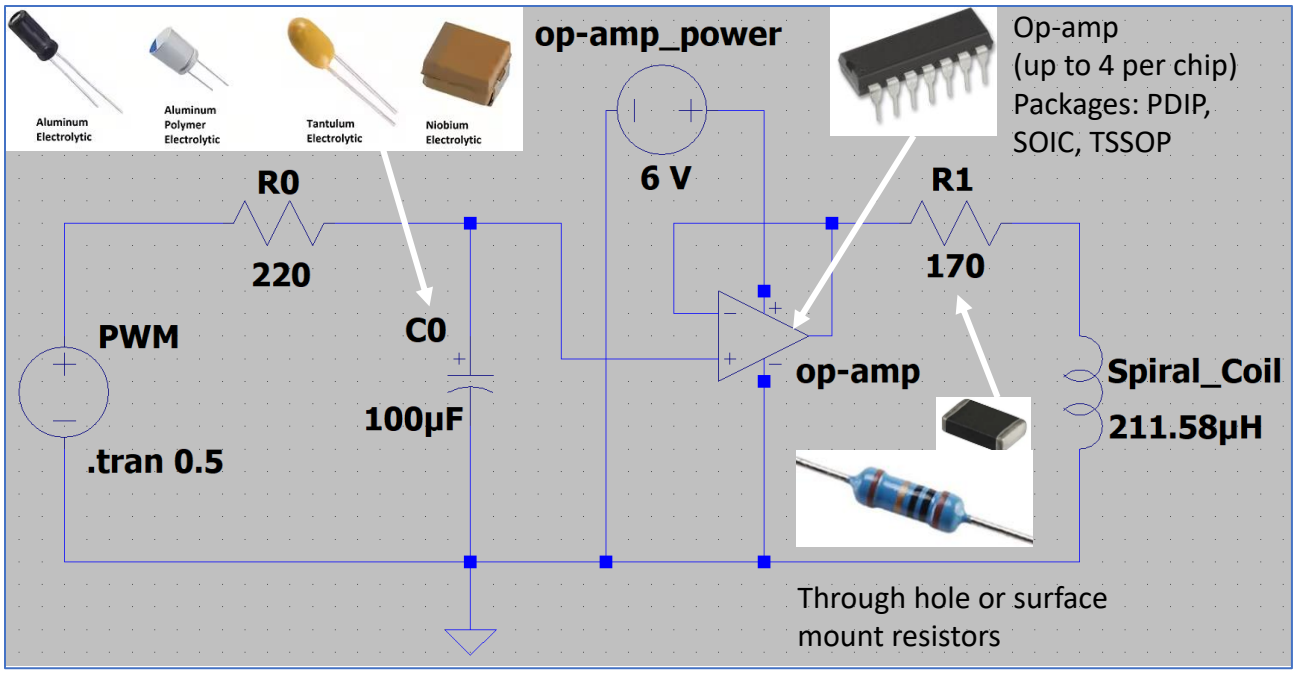


Fig. 13. The schematic diagram of the current driver incorporating PWM, RC filter, and buffer components.

To simulate the current response in the coil, we utilized the LTspice simulator. The transient phenomena linked to capacitor charging, along with the saturation levels of the current, are illustrated in Fig. 14 for two distinct duty cycles. The characteristic time constant of the filter is $\tau = RC$ (22 ms in our case). It is recommended to wait for at least $\sim 6\tau$ to establish a saturation voltage level, indicating that PWMs are relatively slow sources. The PWM frequency ($1/T$) was set at 1.5 kHz. Ensuring that the current saturation levels exhibit minimal ripple in comparison to their constant amplitudes holds significant importance to our application. The driver scheme we proposed in Fig. 13 successfully meets this requirement. Even for a fractional value of mA, the ripple is negligible.

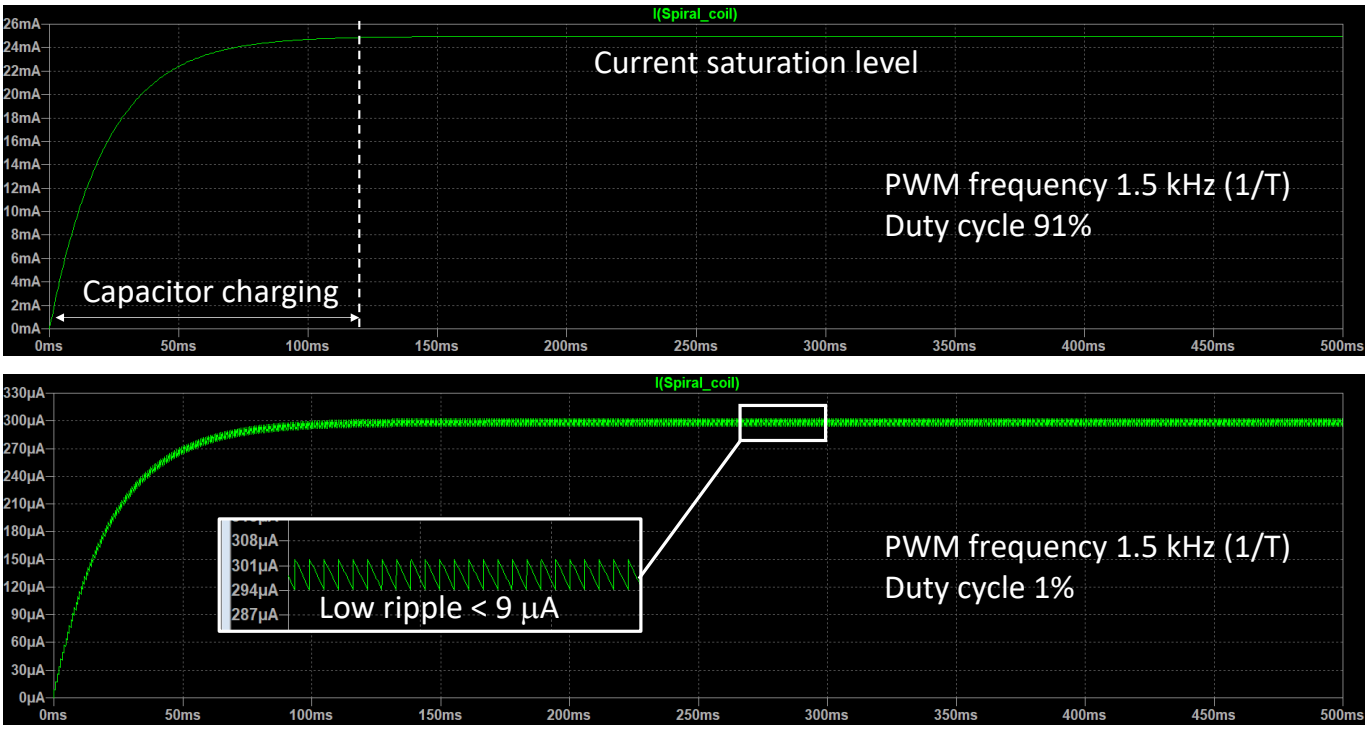


Fig. 14. The coil's current response is examined for two duty cycles within the circuit shown in Fig. 13.

The PWM driver depicted in Fig. 13 was implemented to control two channels, employing the Arduino Mega 2560 R3 microcontroller and the TLV2372 dual operational amplifier. Fig. 15 showcases the photographs and schematic diagram illustrating its principle. Fig. 16 illustrates the pin configuration of Arduino Mega 2560, which is categorized into general digital GPIO and PWM (marked in pink), communication (SPI and I2C), analog input, as well as 3.3V and 5V DC sources. The op-amp was energised from the Arduino's 5V DC source. The maximum achievable PWM frequency on most pins is around 31.25 kHz. We selected a frequency of 1.5 kHz, as it is nearly the upper limit for the multi-channel PWM driver, which we recommend below. However, it is important to note that higher frequencies allow the use of lower capacitance C_0. In digital circuits, it is common practice to incorporate high-resistance shunt resistors (in our case, 1 MΩ) between the analog inputs of the Analog-to-Digital Converter (ADC) and ground. This could be recommended to mitigate the effects of a "floating ground" when the analog input is disconnected. The duty cycle value can be set with an accuracy of up to 8 bits, offering a range of 256 values. ADC provides a resolution of 10 bits, corresponding to 1024 levels with the 5V reference on the board.

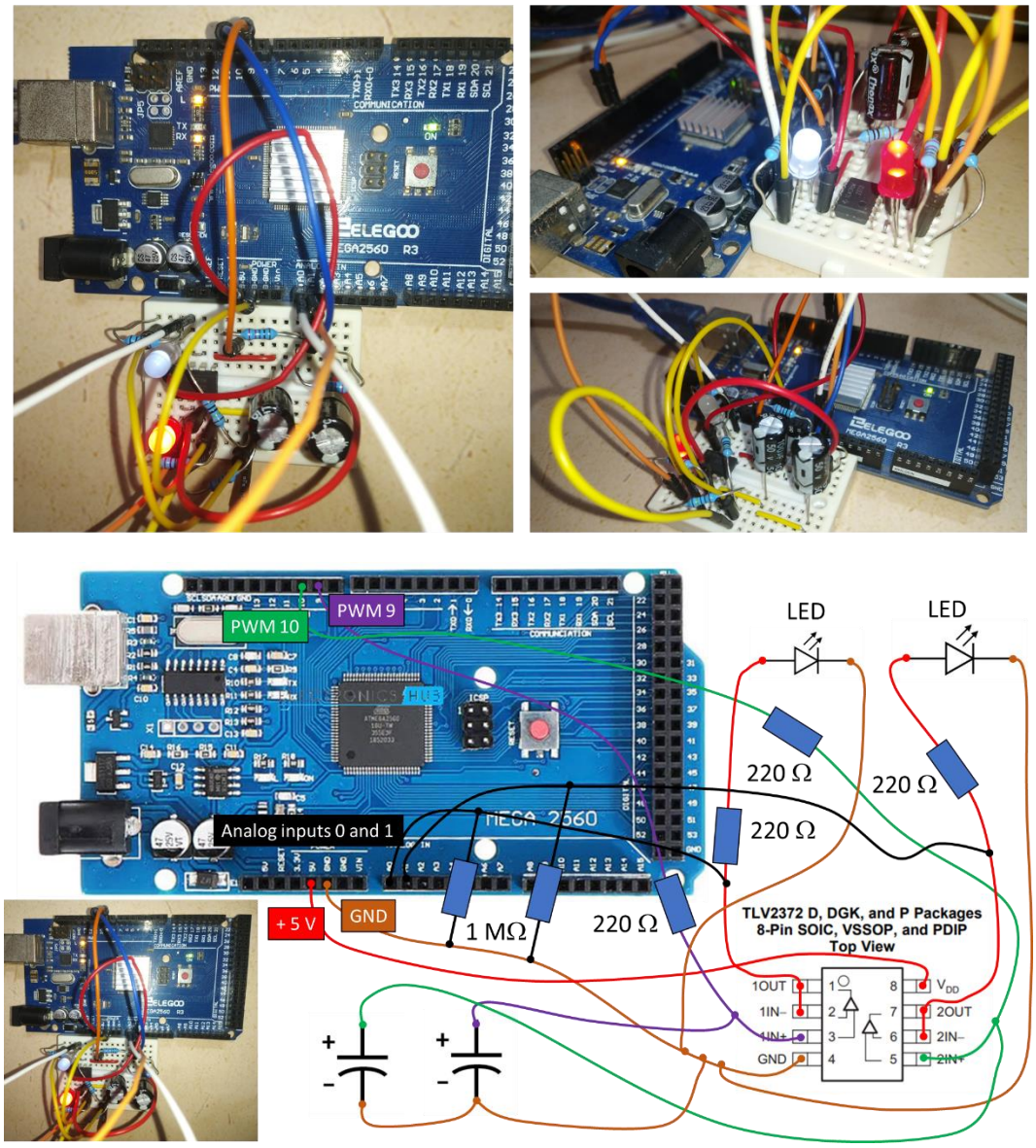


Fig. 15. The PWM driver implemented to control two channels. It is based on the Arduino Mega 2560 R3 microcontroller and the TLV2372 dual operational amplifier.

The performance of the circuit is depicted in Fig. 17. The Arduino program was compiled and uploaded to the microcontroller through its USB port. The analog input data is then read by the Python program to plot the relationship between the output voltages of the two channels and the magnitude of the duty cycle. The TLV2372 series of operational amplifiers belongs to the rail-to-rail type. This design characteristic allows for input and output voltages to span a wide range, extending from ground (0) to the source level (5V in our case). Nevertheless, it's worth noting that amplifier sensitivity tends to be reduced near zero (~1.2% duty cycle) and as values approach the upper limits (~90% duty cycle). Despite this behaviour, these effects do not present any limitations for our applications. This is especially true for the upper voltage level, as it can be readily elevated by making a

minor adjustment to the source voltage, such as increasing it to 6V, for instance. The output voltage can be calculated quite accurately as:

$$V_{out} = \begin{cases} \frac{5\,\mathrm{V}\times(\mathrm{duty\_cycle}-\Delta)[\%]}{100}, & \Delta \le \mathrm{duty\_cycle} < \mathrm{saturation} \\ 0, & \mathrm{duty\_cycle} < \Delta \end{cases} \quad (25)$$

where 5V is the Arduino's default reference voltage and $\Delta \sim 1.2\%$ (this value will vary for each operational amplifier, but it will remain relatively small). The uncertainty in output voltage did not exceed ± 50 mV, equivalent to a current uncertainty of ± 0.3 mA (maximum) through a 170 Ω load (see Fig. 13). We have modelled the impact of this level of uncertainty on the final induction, and it is found to be negligible.

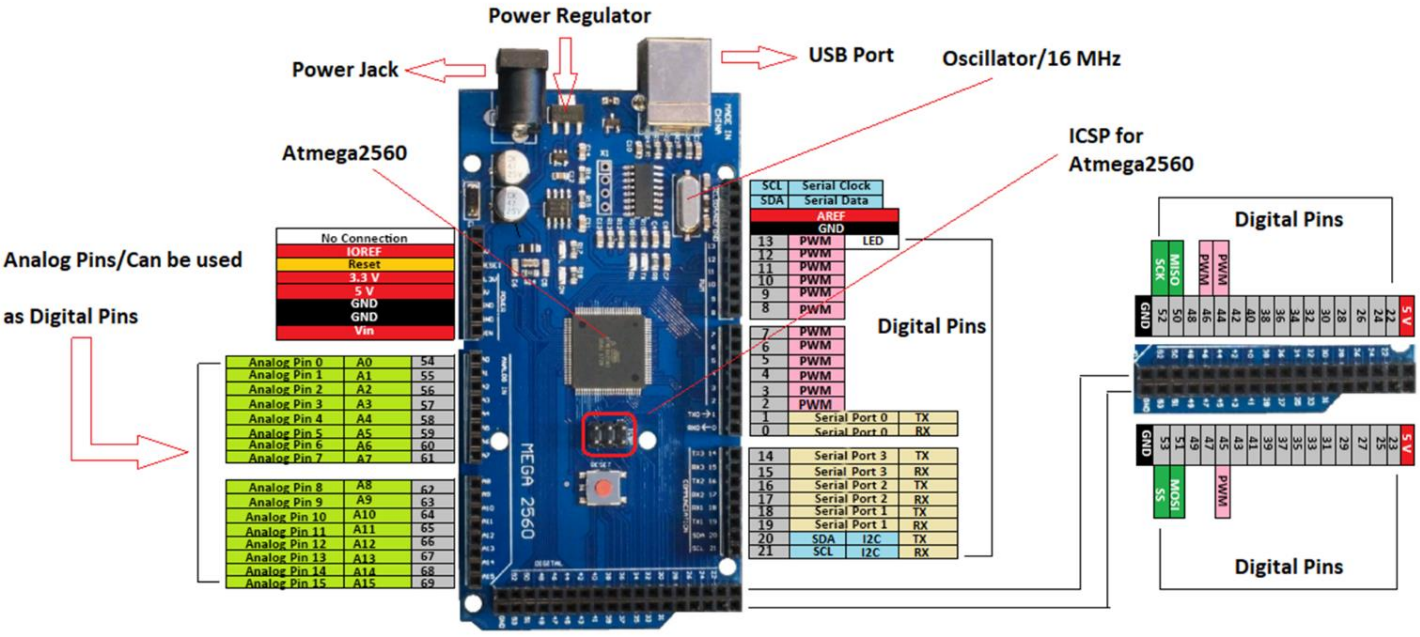


Fig. 16. The pin configuration of Arduino Mega 2560. PWM outputs (15 in total) are marked in pink. The power (voltage) regulator supplies a 5V reference voltage used by all digital microchips and GPIO outputs. It is advisable to attach a small aluminium heat sink (finned or comb-style) to this microchip.

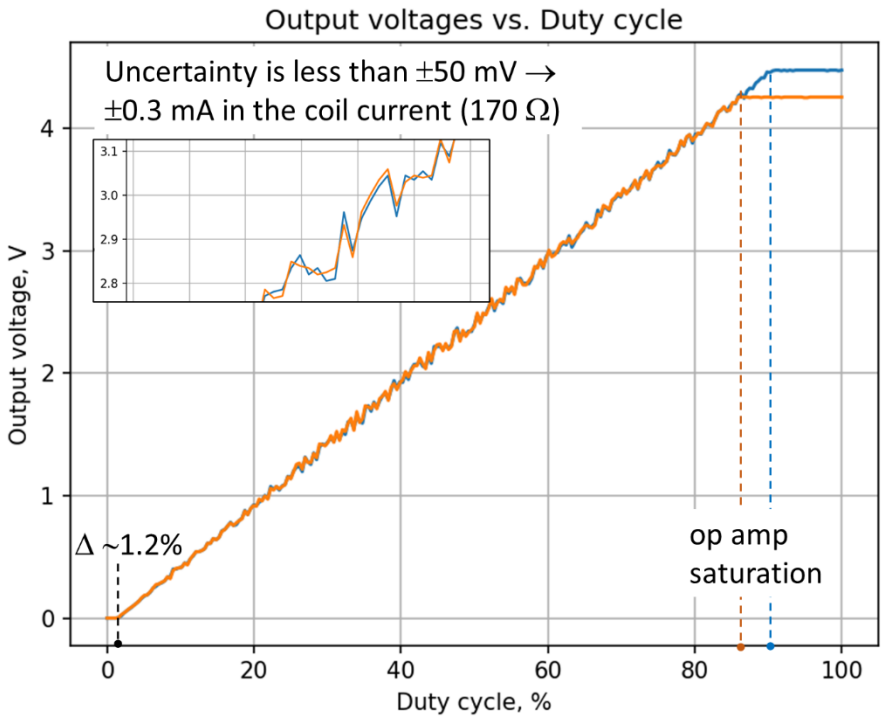


Fig. 17. The performance of the 2-channel PWM driver in Fig. 15.

4.2.2 Multichannel PWM driver with current amplification

The Arduino Mega 2560 board has a total of 15 PWM outputs (see Fig. 16). However, when designing a shimming coil, a significantly larger number of control channels may be required. To address this requirement, the PCA9685 16-channel PWM driver with the I2C interface can be utilized, enabling the addressing of up to 62 boards on a single data bus, as shown in Fig. 18, providing access to a maximum of 992 PWM channels. Notably, each channel allows for individual duty cycle configuration with a high precision of 12 bits resolution. The microchip can be conveniently acquired in a fully assembled state, integrated onto the board along with all the essential peripherals. Each PCA9685 PWM driver, possessing a distinct hardware-configured I2C address, can be seamlessly connected to either a Raspberry Pi (RPi) or an Arduino. These platforms offer dedicated pins that facilitate control over peripheral devices. To exercise direct control over the drivers from a PC (Windows, MAC, or Linux), the utilization of a USB-to-I2C adapter becomes necessary. We suggest using Adafruit MCP2221 breakout. The MCP2221 microchip additionally facilitates GPIO functionality, accommodating components such as buttons and LEDs. Moreover, it supports ADC and digital-to-analog conversion (DAC) capabilities. Consequently, through the utilization of dedicated Python libraries, it becomes feasible to exercise control over these peripherals without necessitating the involvement of microcontrollers, as depicted in Fig. 19.

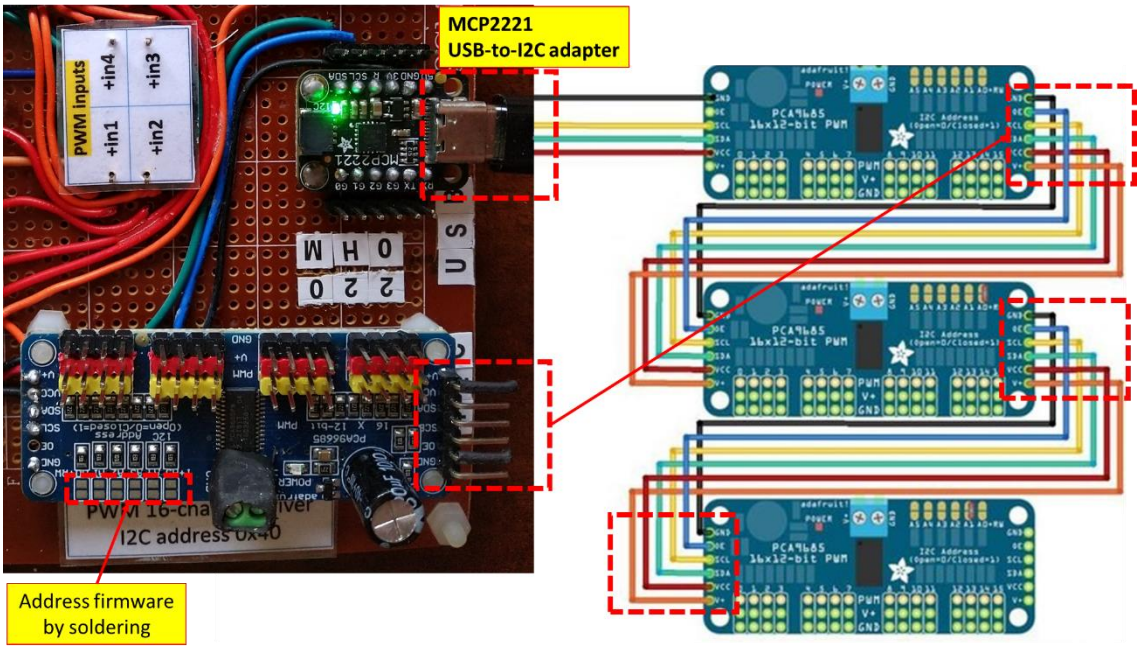


Fig. 18. Up to 62 PCA9685 boards with distinct I2C addresses can be interconnected on a single data bus.

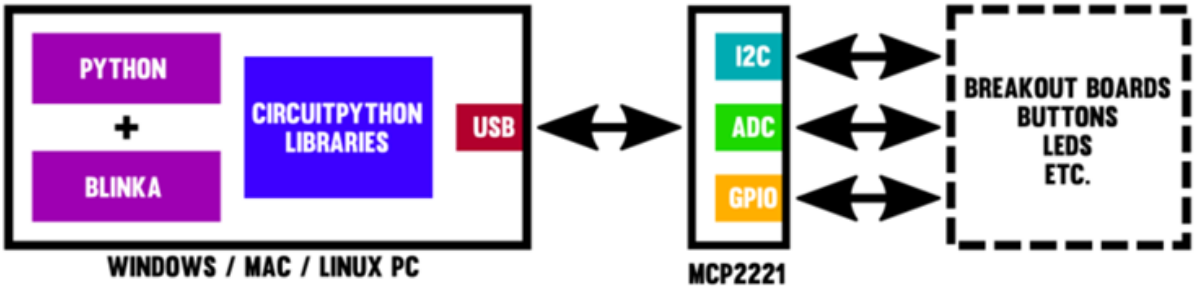


Fig. 19. Using the MCP2221 breakout USB-to-I2C to control peripheral devices on a PC via the I2C communication protocol.

CircuitPython firmware is software written in C for microcontroller boards. It houses the CircuitPython interpreter for executing programs and specific code to manage hardware components like USB, I2C, SPI, GPIO pins, etc., on the microcontroller. CircuitPython Libraries are Python-written modules within the firmware, designed to control microcontroller peripherals and interact with accessories. However, these libraries might be needed on devices lacking CircuitPython, like Windows or macOS. This gap is bridged by Adafruit Blinka. Blinka, a separate software, mimics CircuitPython's hardware control. It provides alternate implementations for native CircuitPython modules like board, busio, and digitalio. This allows Python code to resemble CircuitPython and utilize its libraries, even on non-CircuitPython systems.

By merging the concept of a PWM driver in Fig. 13 with the functionalities of a dedicated PWM source (PCA9685) and a USB-to-I2C adapter (MCP2221), we present a multi-channel driver in Fig. 20, complete with a current booster. In Fig. 21, you can see an implemented driver prototype featuring four channels with current boosters and four spiral coils. A single L7805 5V voltage regulator will suffice for powering all digital circuits in the driver, and likewise, one L7806 6V regulator should be adequate for the buffers due to their low current consumption. However, it is advisable to use one 6V regulator for every four current boosters. The current booster consists of an LM324N quadruple op amp, with each channel's output connected to the bases of NPN BJT transistors. Negative feedback is achieved by monitoring the voltage across the load, which includes a coil and a current limiting resistor, in the emitter circuit. This configuration offers both a high input resistance, unit gain similar to a conventional buffer, and a high output current. We used the popular 2N2222 NPN transistor in a plastic TO-92 package (P2N2222A).

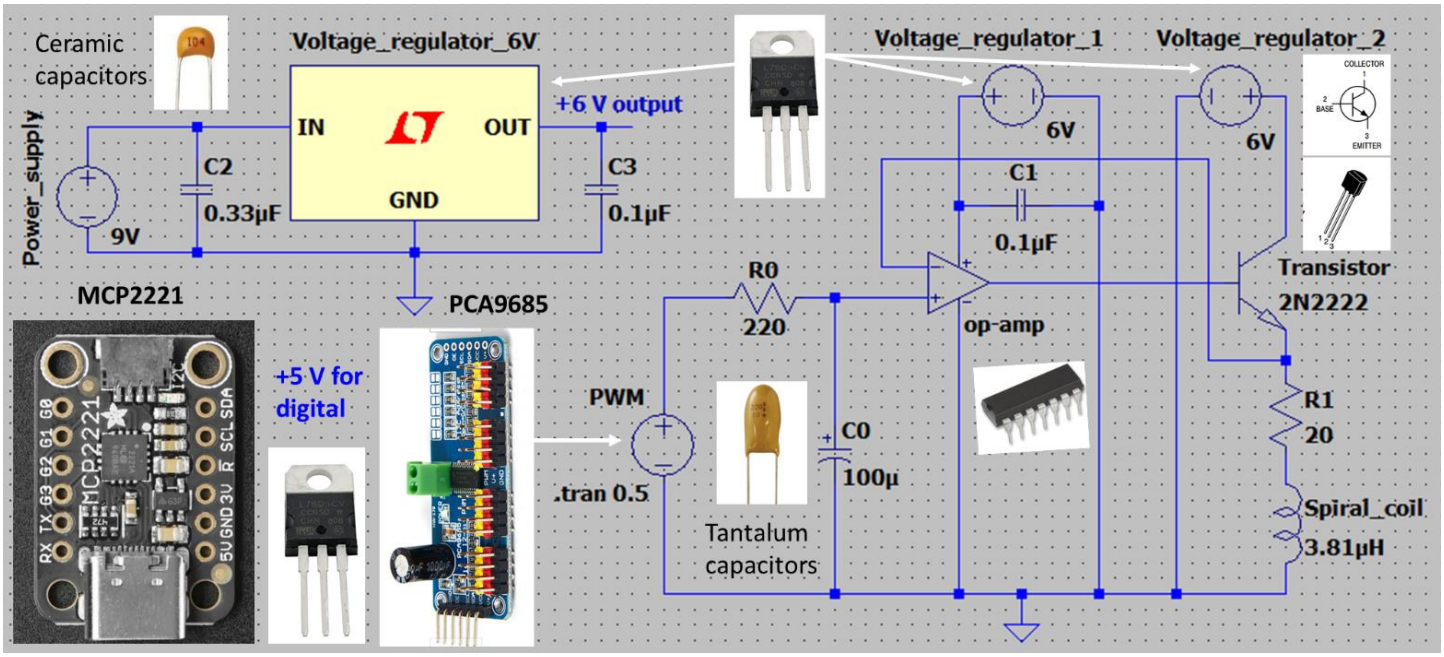


Fig. 20. A multi-channel PWM driver, complete with a current booster.

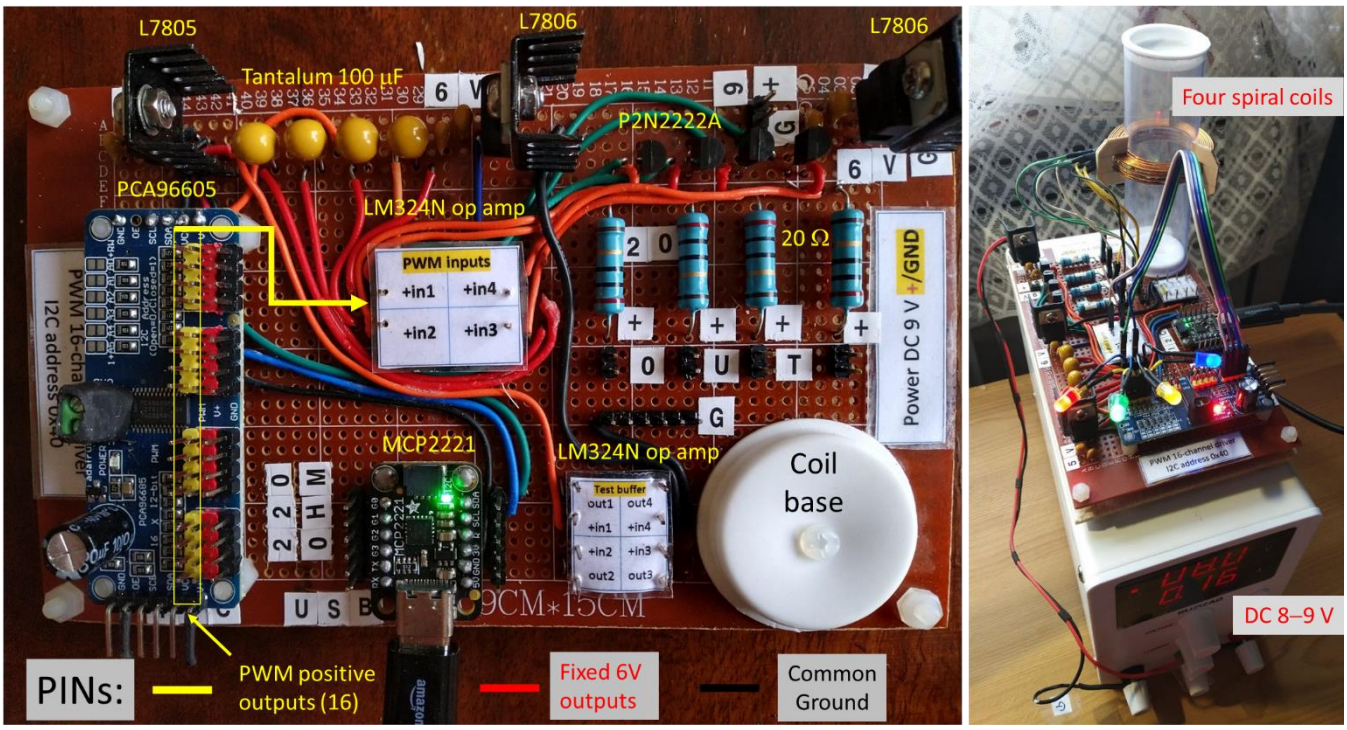


Fig. 21. An implemented driver prototype featuring four channels with current boosters and four spiral coils.

The 600 mAdc collector-emitter current of P2N2222A is more than sufficient for our applications. However, the potential issue lies in heat management. The power P dissipated in the transistor is determined by the product of the voltage drop $(V_{in} - V_{out})$ across the emitter-collector junction and the current $I_{ce} = V_{out}/R_{load}$ passing through it:

$$P = (V_{in} - V_{out}) \times I_{ce} = \frac{(V_{in} - V_{out}) \times V_{out}}{R_{load}} \tag{26}$$

This is a quadratic function by V_{out} that reaches its maximum at $V_{out} = V_{in}/2$:

$$P_{max} = \frac{V_{in}^2}{4R_{load}} \tag{27}$$

If the voltage drop is substantial, the maximum permissible power dissipation may be exceeded, even within the allowable current limits. Therefore, it is advisable to maintain the supply voltage V_{in} at the collector as close as possible to the maximum output voltage $V_{out\_max} = 5\text{V}$. We have selected V_{in} of 6V, which is supplied by a shared 6V voltage regulator for four current boosters. If the coil resistance is low, considering only the current-limiting resistance of 20 Ω, the maximum power dissipation is calculated to be $P_{max} = 0.45$ W.

The junction temperature T_J (℃) in a transistor can be determined according to the thermal equation:

$$T_J = T_A + P_D \times \theta_{JA} \quad (28)$$

where $\theta_{JA} = \theta_{JC} + \theta_{CH} + \theta_{HA}$ is the junction-to-air thermal resistance (℃/W), T_A is the ambient (room) temperature (℃), P_D is the dissipated power (W), θ_{JC} is the junction-to-case thermal resistance (℃/W), θ_{CH} is the case-to-heat sink thermal resistance (℃/W), and θ_{HA} is the heat sink-to-ambient thermal resistance (℃/W). The TO-92 transistor package does not include provisions for a heat sink. According to the datasheet for the P2N2222A, the thermal resistance θ_{JA} is specified as 200 ℃/W. For the maximum dissipated power P_{max} of 0.45 W and an ambient temperature T_A of 25 ℃, we calculate the junction temperature T_J of 115 ℃, which is still below the maximum permissible temperature of 125 ℃ for this type of transistors. We can also use inexpensive transistors, like BD139 with TO-126 package, which permit the attachment of an external heat sink. With a minimum load resistance of 20 Ω, such drivers can provide a maximum current of V_{out_max}/R_{load}, which equals 250 mA (5V/20). It is important to consider the power dissipated in the current-limiting resistor, which can reach a maximum of 1.25 W ($V_{out\_max}^2/R_{load}$). To accommodate this, we selected resistors with a power rating of 2 W.

Heat management for voltage regulators follows a similar approach. These regulators provide fixed output voltages, and the input voltage should be approximately 2 V higher than the output voltage. In our driver, we use L78-series positive 5V and 6V voltage regulators with TO-220 package. Therefore, the DC power source must be at least 8V. It is easy to find a standard DC power supply with an output voltage of 9V. The most critical aspect is the management of the 6V regulators that power the current boosters: one regulator for every four boosters. For every four boosters, we calculate the maximum power dissipated by the regulator according to Eq. (26): $(9\text{V} - 6\text{V}) \times 0.25\text{A} \times 4 = 3$ W. For TO-220 package, $\theta_{JC} = 5$ ℃/W. By using a heat paste, as shown in Fig. 22, we can reduce θ_{CH} to negligibly small values. We employed a natural convection heat sink with $\theta_{HA} \approx 24$ ℃/W. Therefore, using Eq. (28), we calculate the maximum T_J as follows: $T_J = 25 + 3 \times (24 + 5) = 112$ ℃. This temperature is still below the maximum T_J for L78-series, which is 125 ℃. By employing more efficient heat sinks, whether through natural convection or with fan assistance, it is possible to significantly reduce the junction temperature. You can familiarize yourself with the principle of forced cooling in Appendix.

The total current consumption depends on the number of channels, N. In our case, it can be calculated as $(N \times 0.25)$A. Therefore, a DC power supply with a capacity greater than $(9 \times N \times 0.25)$W will be required.

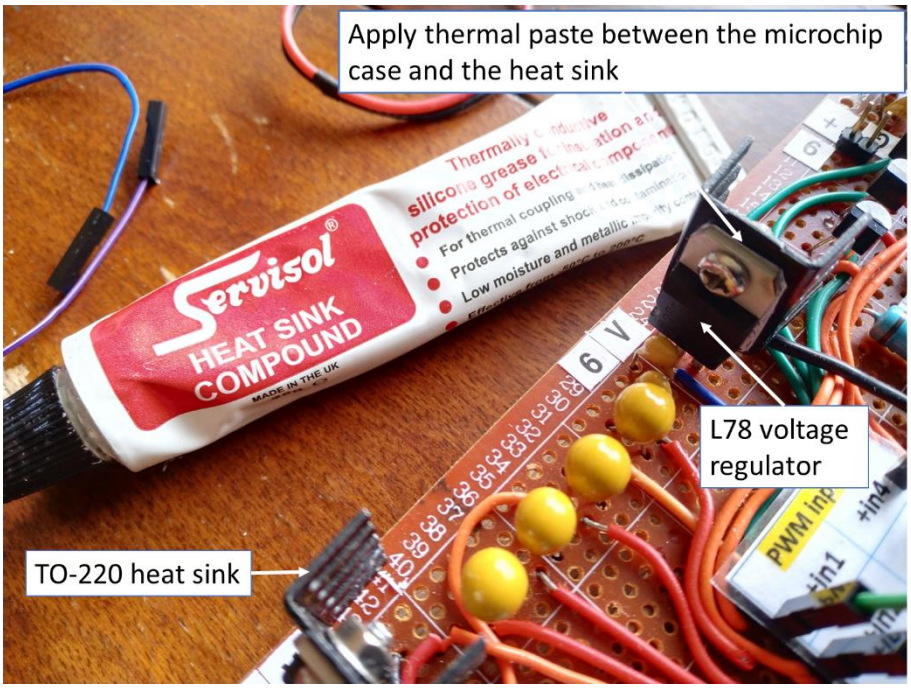


Fig. 22. Methods of heat management employed in the driver.

The graphical user interface (GUI) for controlling the electronic driver in Fig. 21 is depicted in Fig. 23. This interface was developed using Python's Tkinter library. The entire program code is archived in Appendix for reference. The first four channels are designated for controlling the coils, while the last four channels are allocated for controlling LEDs, illustrating the operational principle of the PWM driver, where brightness is contingent on the duty cycle. Upon setting individual duty cycle values within the range of $(0-1)\times 100\%$, they can be transferred to PCA9685 by selecting the "Apply Settings" virtual button. Subsequently, the program can be closed, as the duty cycle values will be retained within the microchip as long as power is supplied.

The driver output characteristics displayed in Fig. 23 were acquired from the four channels through the utilization of the same testing program implemented with Arduino Mega. Output curves associated with current boosters exhibit good repeatability and linearity, and minimal ripple. However, saturation arises at an earlier stage, specifically at ~70% duty cycle, and the dead zone (see Eq. (25)) at low duty cycle values is more pronounced compared to Fig. 17. These effects are attributed to the use of the LM324N op amp that lacks rail-to-rail performance. While these limitations may not be critical for the prototype, they necessitate consideration when designing a practical circuit.

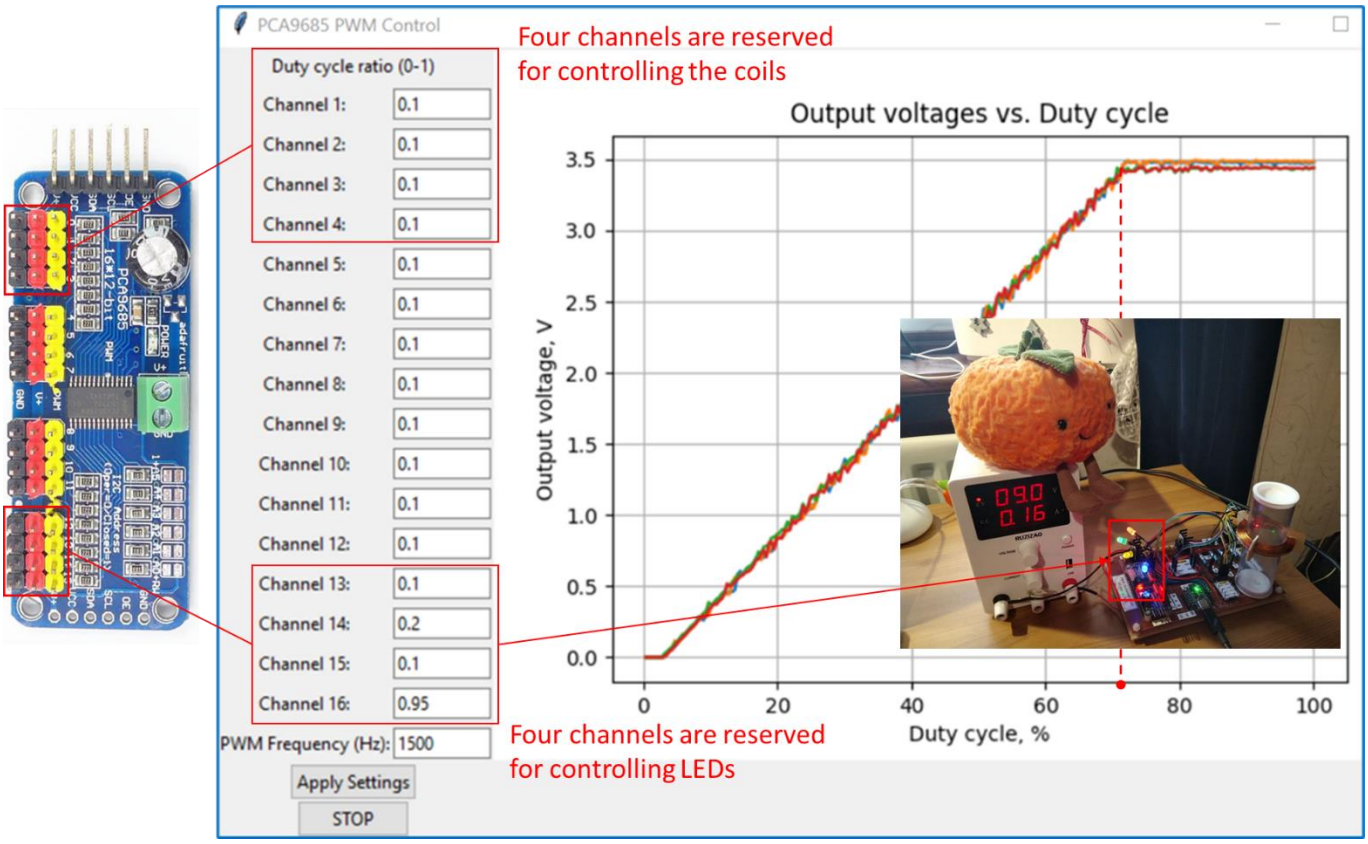


Fig. 23. GUI for controlling the electronic driver in Fig. 21.

The functional blocks within the driver (refer to Fig. 21) are intentionally not permanently interconnected. This design choice offers greater flexibility, allowing for individual learning of each block. The circuit can be configured for various types of measurements, and users are encouraged to experiment with microchips and devise their own measurement scenarios.

4.3 Shimming coil optimisation for the number of turns

Newton's method, which we have utilized in our project, offers the advantage of not imposing any constraints on the optimization parameters. Three additional gradient-based optimization methods from the Python optimisation library, which will be reserved for the turns optimization, are also unconstrained: "trust-exact", "trust-krylov", and "trust-ncg". However, when it comes to optimization by turns, an initial constraint arises: $\theta_{2m} > \theta_{1m}$. Furthermore, our observations from optimizing the current profile reveal that its values can be bipolar. Consequently, when applying a fixed current and optimizing by turns, it becomes evident that the spiral coils can be wound in either direction. Nonetheless, in Eq. (6), the quantity $\gamma\theta$, which determines the radius, cannot have a negative value. Therefore, it is evident that modifications are necessary for the functional in Eq. (17) in order to eliminate the aforementioned constraints.

Firstly, let us express θ_{2m} in terms of a new variable χ_m, which can take both positive and negative values:

$$\theta_{2m}(\chi_m) = \theta_{1m} + \chi_m \text{sign}(\chi_m) \tag{29}$$

where $\theta_{1m} = 2\pi R/d - \pi = const$ for all coils (recall Figs. 7 and 8) and $\text{sign}(\chi)$ is a sign function. In this case, the entire second term in Eq. (29) maintains a positive value, which signifies the existence of extra turns relative to θ_{1m}. The hyperbolic tangent is a viable candidate for a differentiable sign function:

$$\text{sign}(\chi) = \tanh(a\chi) \tag{30}$$

where a is a positive constant that governs the speed of transition between positive and negative unit values of the hyperbolic tangent function. Now, the functional in Eq. (17) that is intended to be optimized by the number of turns will be reformulated as follows:

$$J(\vec{\boldsymbol{\chi}}) = \sum_{q=1}^{Q} \left(B_z\left(z_q, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\chi}}, \vec{\mathbf{I}}\right) + B_0\left(z_q\right) \right)^2 \tag{31}$$

where

$$B_z\left(z_q, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\chi}}, \vec{\mathbf{I}}\right) = \frac{\mu_0 I}{4\pi\gamma} \sum_{m=1}^{M} \text{sign}(\chi_m) \left(\ln\left(\frac{2\gamma\theta_{2m}(\chi_m)+d+\sqrt{(2\gamma\theta_{2m}(\chi_m)+d)^2+4(z_q-z_m)^2}}{2\gamma\theta_{1m}+d+\sqrt{(2\gamma\theta_{1m}+d)^2+4(z_q-z_m)^2}} \right) + \right.$$
$$\left. + \frac{2\gamma\theta_{1m}+d}{\sqrt{(2\gamma\theta_{1m}+d)^2+4(z_q-z_m)^2}} - \frac{2\gamma\theta_{2m}(\chi_m)+d}{\sqrt{(2\gamma\theta_{2m}(\chi_m)+d)^2+4(z_q-z_m)^2}} \right) \tag{32}$$

The induction from an individual spiral coil can now exhibit a change in sign based on the winding direction (left or right), which is determined by the sign of χ_m. For the gradient vector and the Hessian matrix, we obtain:

$$\frac{\partial J(\vec{\boldsymbol{\chi}})}{\partial \chi_m} = 2 \sum_{q=1}^{Q} \left(B_z\left(z_q, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\chi}}, \vec{\mathbf{I}}\right) + B_0\left(z_q\right) \right) \frac{\partial B_z\left(z_q, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\chi}}, \vec{\mathbf{I}}\right)}{\partial \chi_m} \tag{33}$$

$$\frac{\partial^2 J(\vec{\boldsymbol{\chi}})}{\partial \chi_m \partial \chi_n} = 2 \sum_{q=1}^{Q} \frac{\partial B_z\left(z_q, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\chi}}, \vec{\mathbf{I}}\right)}{\partial \chi_m} \times \frac{\partial B_z\left(z_q, \vec{\boldsymbol{\theta}}_1, \vec{\boldsymbol{\chi}}, \vec{\mathbf{I}}\right)}{\partial \chi_n} \tag{34}$$

Performing analytical calculations of the partial derivatives in Eqs. (33), (34) would be highly cumbersome. Fortunately, there is a Python library called SymPy that enables symbolic calculations. With the help of this library and simple code (see Appendix), we calculated the partial derivative symbolically and implemented the corresponding function in the code.

The optimization algorithm for the number of turns is detailed in the Appendix. Its GUI is similar to that used for the current optimization but with some notable changes. The outer diameter of each coil is now a variable subject to optimization. Additionally, as illustrated in Fig. 24, three new parameters have been introduced: the hyperbolic tangent constant a in Eq. (30), the initial number of turns N (which can be positive or negative), and the utilization of four gradient unconstrained methods.

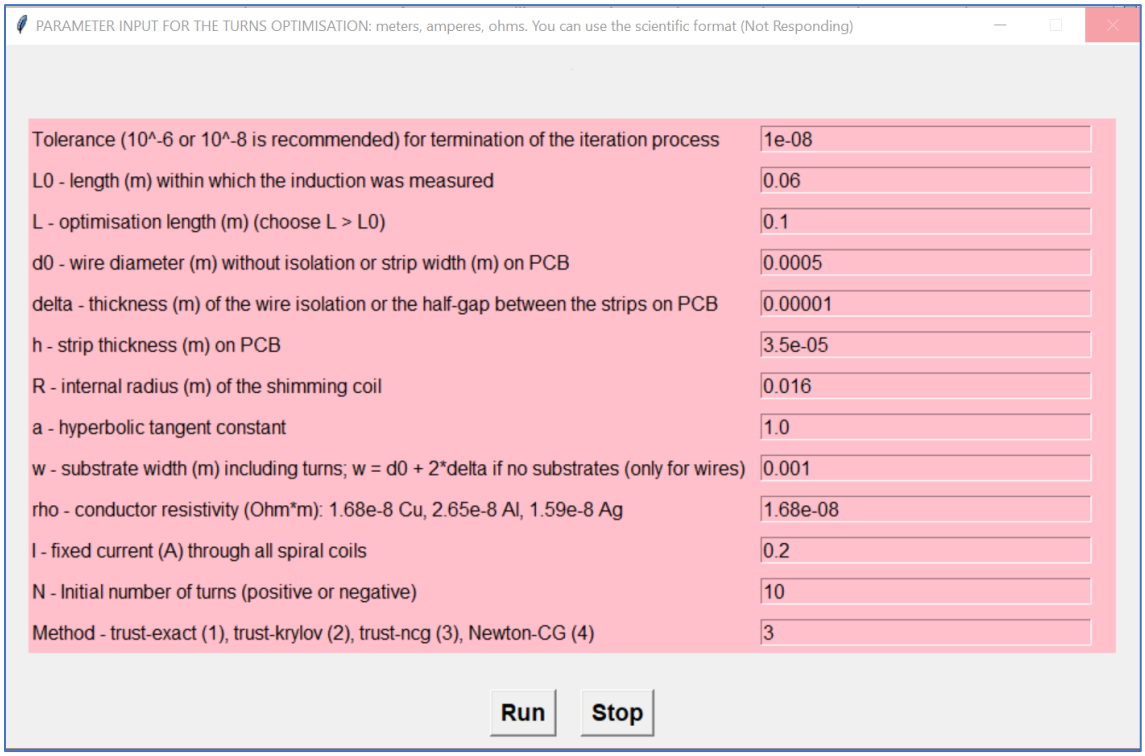


Fig. 24. GUI for the turns optimisation algorithm.

As with the current optimization, there is not just one, but an entire range (“valley”) of acceptable solutions. The specific solution that the optimization converges to depends on the choice of wire diameter or strip width, isolation thickness, initial values for turns, current strength, hyperbolic constant, and optimization method (now there are four to choose from). If the hyperbolic constant is selected too large, the program may report an overflow (red messages in console) but will continue to execute. Therefore, when searching for the optimal coil profile, you will need to experiment with all these parameters. It is interesting to note that the Newton-CG method

performed less effectively than other gradient methods, which is why we have ranked it as number four in the GUI window.

As an illustration, in Fig. 25, we present the results of numerical optimization for a shimming coil (wire or PCB strip) with the following design parameters:

```
 1. total current through all spiral coils = 0.2 A
 2. strip thickness = 3.5e-05 m
 3. conductor diameter/width without isolation/gap = 0.0005 m
 4. total conductor diameter/width = 0.000521 m
 5. total substrate width, including the wire diameter = 0.001 m
 6. residual induction length = 0.06 m
 7. total optimisation length = 0.1 m
 8. adjusted internal radius = 0.01586 m
 9. total length of the spirals = 116.77919800287563 m
10. total wire resistance = 9.991826402861111 Ohms
11. total strip resistance = 112.10803008276069 Ohms
12. power dissipated in the whole wire stack = 0.39967305611444454 W
13. power dissipated in the whole strip stack = 4.4843212033104285 W
```

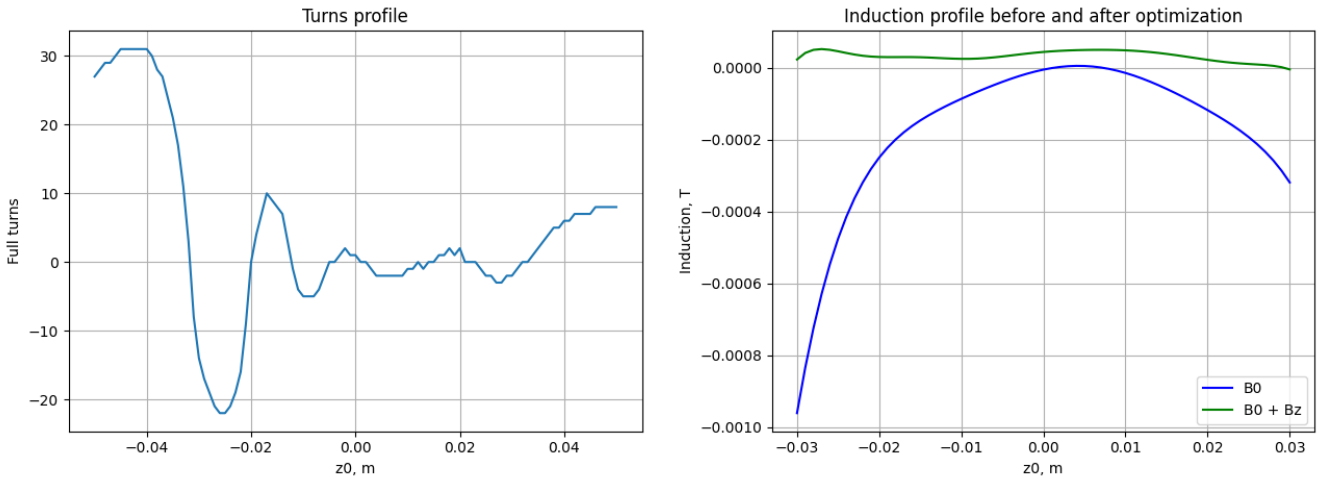


Fig. 25. An example of the turns optimisation.

The optimisation offers a fairly complex turns profile, which results in perfect residual field compensation performance: while the field is not entirely eliminated, it remains almost constant along the entire axis, which is precisely the desired outcome for shimming. Each spiral coil contains only the full number of turns, which makes them easier to manufacture. All coils are connected in series and have the same current I. The driver will be a low-power single-channel DC voltage source, which, of course, is much simpler than the driver used for the current optimization. However, this approach sacrifices flexibility for fine-tuning the field once the sample is in place.

Some optimisation algorithms offer the options of utilizing numerical derivatives for both the gradient and the Hessian. However, the explanation of the options for numerical derivatives as presented in the Python optimisation library manual can be rather confusing:

> *The gradient and the Hessian may be approximated using three finite-difference schemes: {'2-point', '3-point', 'cs'}. The scheme 'cs' is, potentially, the most accurate but it requires the function to correctly handle complex inputs and to be differentiable in the complex plane. The scheme '3-point' is more accurate than '2-point' but requires twice as many operations.*

Let us recall the standard definitions for numerical first and second derivatives:

$$\frac{df}{dx} \approx \frac{f(x+h)-f(x-h)}{2h} \tag{35}$$

$$\frac{d^2f}{dx^2} \approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2} \tag{36}$$

where h is a small displacement near x. Based on these equations, it is evident that the first derivative is calculated using the two-point scheme, while the second derivative is determined through the three-point scheme. However, the manual also mentions a three-point scheme for the first derivative and a two-point scheme for the second. It is not clear what that means. Most likely, the manual suggests using the '2-point' scheme for calculating the first derivative and the '3-point' scheme for the second derivative.

They do not provide any explanation for what 'cs' stands for. However, it appears to be a fascinating technique known as the *"complex step"*, which has gained significant attention in various domains of numerical analysis, particularly in solving differential equations. For further exploration, you may refer to the following references: C. Corner, R. Abreu et al., N. Higham, and MDO Lab. The primary advantage of employing the complex step method lies in its calculation stability, where the accuracy remains intact even as the step size decreases. This stability ensures reliable results and allows for precise computations without sacrificing accuracy when using smaller steps. To apply the complex step method for calculating the derivative, it is essential that the function possesses a unique analytic continuation to the complex plane. While we have not extensively explored this method, it is likely that a univalent Riemann surface is required for the function's range. In our case, the function might be too complicated to straightforwardly employ this method, necessitating potential modifications to make it applicable. Firstly, the standard logarithm must be expressed in complex form, considering both the logarithm of the modulus and the complex part of the phase. Additionally, resolving the issue of calculating roots in the complex plane will be crucial, as its Riemann surface is two-sheeted, introducing potential ambiguities in the process.

Appendix

Solenoid calculator

Project folder: "Solenoid_Python". The program has been designed as an interactive console application, providing flexibility for customization and accepting parameters as constants within the program code. This can be accomplished by uncommenting lines 25-31 and commenting out lines 14-22. The results of the field distribution calculation are stored in the "Solenoid_Field.txt" file in a format with three columns: z0 (m), H (A/m), and H (Oe).

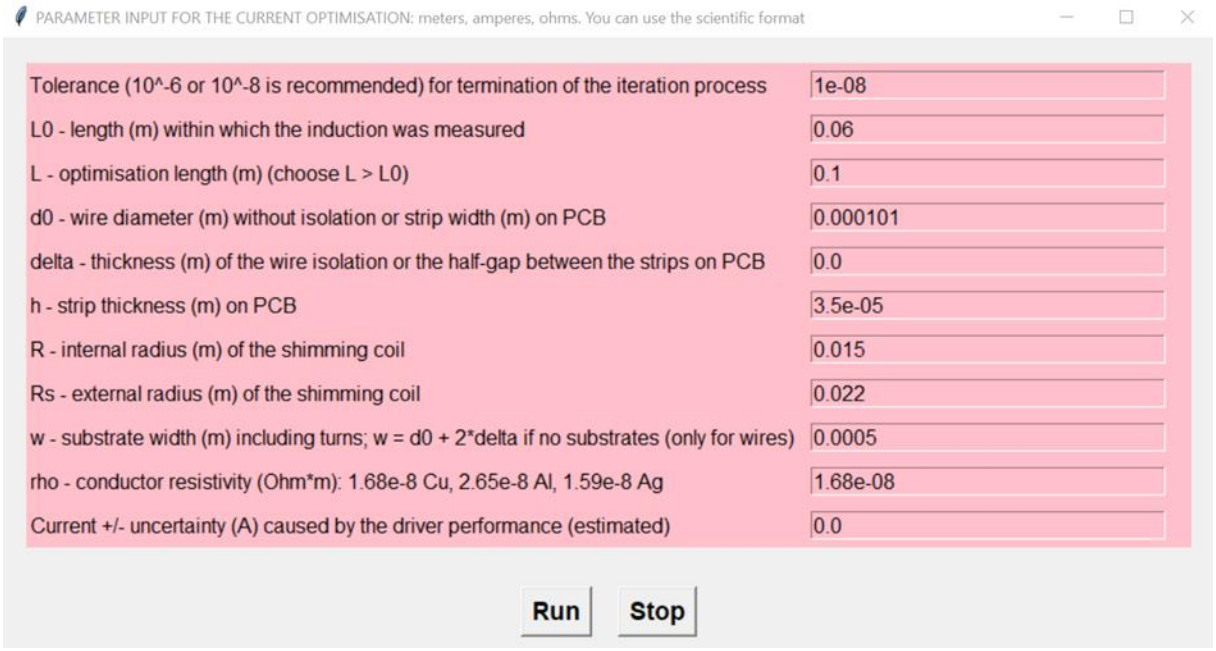
main.py:

```
 1. #
 2. # Solenoid calculator
 3. # Calculation of the magnetic field strength H (A/m and Oe) directed and distributed along the axis of the
solenoid.
 4. #
 5. # Yujie Zhao, University of St. Andrews, Scotland
 6. # 27.04.2023
 7. #
 8.
 9. import numpy as np
10. import matplotlib.pyplot as plt
11. pi = np.pi  # 3.1415...
12.
13. # Console application:
14. print("")
15. print("The solenoid is symmetrical about 0: z0 = [-maxz0, maxz0]")
16. L = float(input("Enter the solenoid length L in meters: "))
17. R = float(input("Enter the solenoid internal radius R in meters: "))
18. d0 = float(input("Enter the wire diameter d0 without isolation in meters: "))
19. delta = float(input("Enter the wire isolation thickness in meters: "))
20. M = int(input("Enter the number M of winding layers: "))
21. maxz0 = float(input("Enter the range maxz0 for z0 = [-maxz0, maxz0] in meters: "))
22. I = float(input("Enter the current I in amperes: "))
23.
24. # Solenoid and wire geometrical parameters:
25. #I = 1  # current in amperes
26. #R = 1.0e-2  # solenoid internal radius in meters
27. #L = 2.0e-1  # solenoid length in meters
28. #M = 6  # number of winding layers
29. #d0 = 1.0e-3  # wire diameter without isolation in meters
30. #delta = 0.0  # wire isolation thickness in meters
31. #maxz0 = L/2.0  # z0 = [-maxz0,+maxz0] is the range of the z-coordinate along the solenoid axis
32.
33. # Derived parameters:
34. d = d0 + 2 * delta  # total wire diameter, including isolation
35. N = int(L / d) + 1  # number of turns in each layer
36. K = N  # number of points of the z-coordinate used for drawing the magnetic field strength
37. gamma = d / (2.0 * pi)  # m/rad is the rotation-to-displacement coefficient
38. psi = pi * N  # 2 * psi is the total rotation angle for all turns in rad
39.
40. # Magnetic field strength H measured in A/m along the solenoid axis as a function of z0
41. def Hz(z0):
42.     value = 0.0
43.     for m in range(1, M + 1):
44.         value = value + (z0 + gamma * psi) / ((R + d * (2 * m - 1) / 2)**2 + (z0 + gamma * psi)**2)**0.5
45.         value = value - (z0 - gamma * psi) / ((R + d * (2 * m - 1) / 2)**2 + (z0 - gamma * psi)**2)**0.5
46.     return value * I / (4.0 * pi * gamma)
47.
48. dz = 2 * maxz0 / K  # coordinate step
```

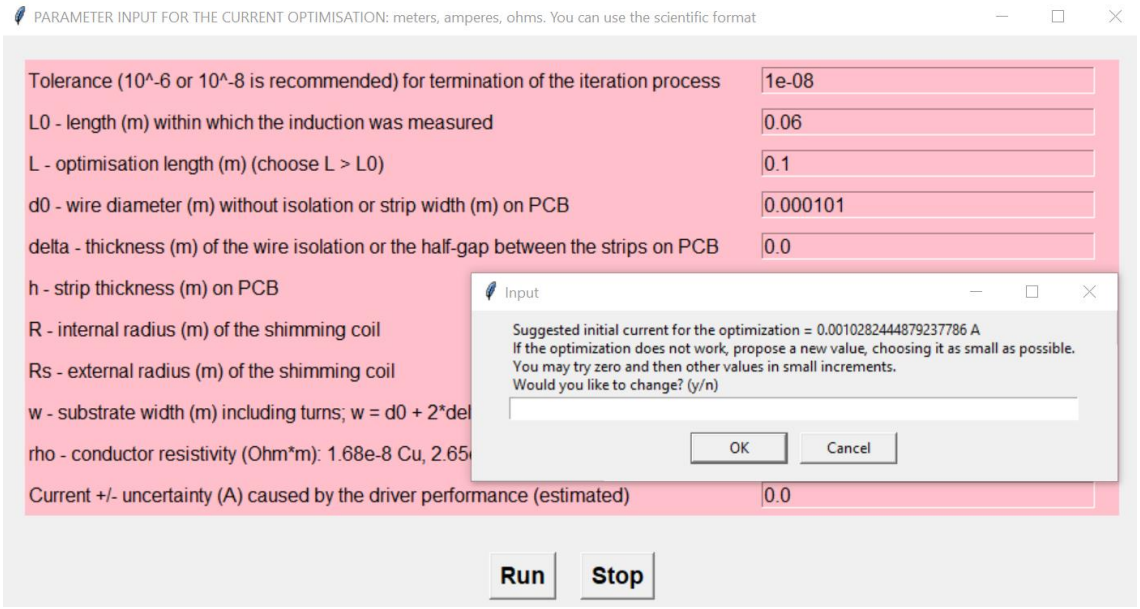
```
z = []  # list of the coordinate points along the solenoid axis in m
H1 = []  # list of the magnetic field points in A/m
for i in range(K + 1):
    z0 = -maxz0 + dz * i
    z.append(z0)
    H1.append(Hz(z0))

z = np.array(z)  # converting into an array
H1 = np.array(H1)  # converting into an array
H2 = (H1 * 4.0 * pi) / 1000  # array of the magnetic field points in Oe

stacked_array = np.column_stack((z, H1, H2))
header = "z0 (m), H (A/m), H (Oe)\n"
with open("Solenoid_Field.txt", "w") as f:
    f.write(header)
    np.savetxt(f, stacked_array, delimiter=",")

# Create two subplots side by side
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

# Plot the first graph (z, H1)
ax1.plot(z, H1)
ax1.set_xlabel("z0 (m)")
ax1.set_ylabel("Hz (A/m)")
ax1.grid(True)

# Plot the second graph (z, H2)
ax2.plot(z, H2)
ax2.set_xlabel("z0 (m)")
ax2.set_ylabel("Hz (Oe)")
ax2.grid(True)

# Display the plots
plt.show()
```

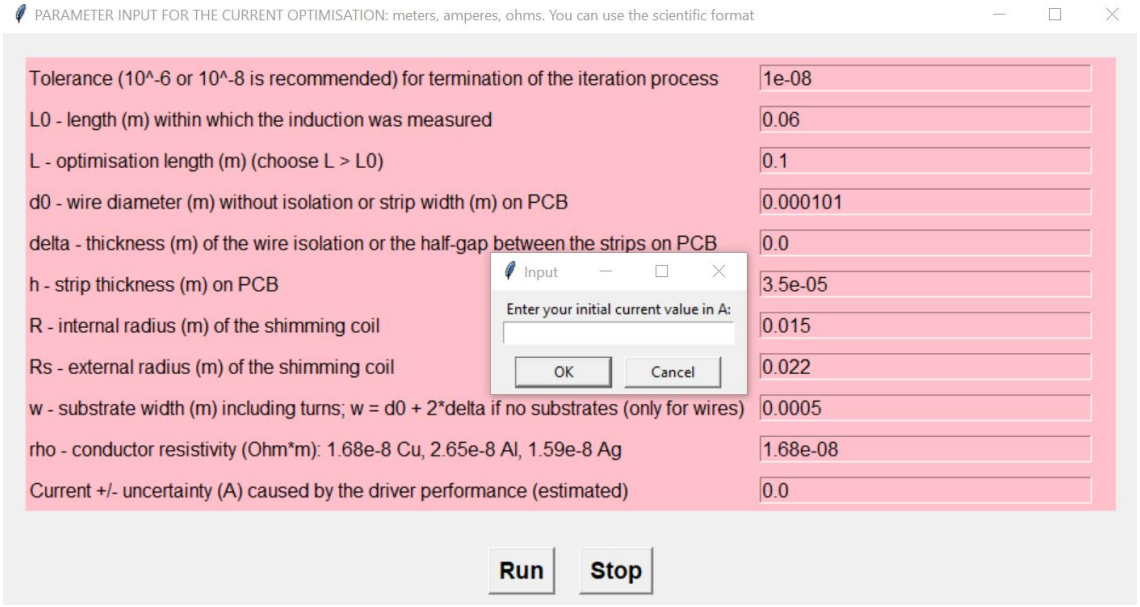
Current optimisation algorithm for the shimming coil

Project folder: "Current_Optimisation_Python". The program features a graphical user interface (GUI) that can be launched by executing the "main.py" file. Other modules in the program project: "NewtonCG.py" (current optimization algorithm) and "ResidualB.py" (distribution of the residual induction). Within the folder, there is a "parameters.txt" log file that should not be removed. This file retains the default parameters for use in subsequent program launches. The file "design_parameters_for_currents.txt" holds the coil's design parameters, which are obtained through user interaction and optimization. The calculated current profiles and corresponding fields are stored in CSV files with self-explanatory names.



The outcome of the optimisation process is highly influenced by the initial values of the currents (see Eq. (23)). To facilitate this, the program prompts two dialog boxes for user input.


PARAMETER INPUT FOR THE CURRENT OPTIMISATION: meters, amperes, ohms. You can use the scientific format
Tolerance (10^-6 or 10^-8 is recommended) for termination of the iteration process
1e-08
L0 - length (m) within which the induction was measured
0.06
L - optimisation length (m) (choose L > L0)
0.1
d0 - wire diameter (m) without isolation or strip width (m) on PCB
0.000101
delta - thickness (m) of the wire isolation or the half-gap between the strips on PCB
0.0
h - strip thickness (m) on PCB
R - internal radius (m) of the shimming coil
Rs - external radius (m) of the shimming coil
w - substrate width (m) including turns; w = d0 + 2*del
rho - conductor resistivity (Ohm*m): 1.68e-8 Cu, 2.65
Current +/- uncertainty (A) caused by the driver performance (estimated)
0.0
Input
Suggested initial current for the optimization = 0.0010282444879237786 A
If the optimization does not work, propose a new value, choosing it as small as possible.
You may try zero and then other values in small increments.
Would you like to change? (y/n)
OK
Cancel
Run
Stop


PARAMETER INPUT FOR THE CURRENT OPTIMISATION: meters, amperes, ohms. You can use the scientific format
Tolerance (10^-6 or 10^-8 is recommended) for termination of the iteration process
1e-08
L0 - length (m) within which the induction was measured
0.06
L - optimisation length (m) (choose L > L0)
0.1
d0 - wire diameter (m) without isolation or strip width (m) on PCB
0.000101
delta - thickness (m) of the wire isolation or the half-gap between the strips on PCB
0.0
h - strip thickness (m) on PCB
3.5e-05
R - internal radius (m) of the shimming coil
0.015
Rs - external radius (m) of the shimming coil
0.022
w - substrate width (m) including turns; w = d0 + 2*delta if no substrates (only for wires)
0.0005
rho - conductor resistivity (Ohm*m): 1.68e-8 Cu, 2.65e-8 Al, 1.59e-8 Ag
1.68e-08
Current +/- uncertainty (A) caused by the driver performance (estimated)
0.0
Input
Enter your initial current value in A:
OK
Cancel
Run
Stop

main.py:

```
#
# The main module for GUI that calls the optimisation algorithms
# Yujie Zhao, University of St. Andrews, Scotland, 30th July 2023.
#

import os
from NewtonCG import *

PARAMETERS_FILE = "parameters.txt"
NUM_PARAMETERS = 11

def stop_main():
    quit()

def run_main():
    # This function will be called when the "Run" button is clicked
    # You can access the parameter values here and call your own function using the parameters

    # Retrieve the parameter values from the input fields
    params = [float(entries[i].get()) for i in range(1, NUM_PARAMETERS+1)]

    # Save the parameter values to a file
    save_parameters(params)

    # params[0] tolerance: tolerance for termination of the iteration process.
    # params[1] L0: length (m) within which the induction was measured
    # params[2] L: length (m) of the optimisation (we suggest L > L0)
    # params[3] d0: wire diameter (m) without isolation or strip width
    # params[4] delta: thickness of the wire isolation or the half-gap between the strips (m) on PCB
    # params[5] h: strip thickness (m) on PCB
    # Both radii will be slightly adjusted: R = d * int(R/d) + d/2 and Rs = d * int(Rs/d) + d/2, where d = d0
+ 2*delta
    # params[6] R: internal radius (m) of the shimming coil
    # params[7] Rs: external radius (m) of the shimming coil
    # params[8] w: substrate width (m); w = d0 + 2*delta if no substrates
    # params[9] rho: conductor resistivity (Ohm*m)
    # params[10] Inoise: current uncertainty (+/-) in A caused by the driver performance

    CurrentOptimisation(params[0], params[1], params[2], params[3], params[4], params[5], params[6],
params[7],
                        params[8], params[9], params[10])

def save_parameters(params):
    # Save the parameter values to a file
    with open(PARAMETERS_FILE, "w") as file:
        for param in params:
            file.write(f"{param}\n")

def load_parameters():
    # Load the parameter values from the file, if it exists
    if os.path.exists(PARAMETERS_FILE):
        with open(PARAMETERS_FILE, "r") as file:
            lines = file.readlines()
            for i, line in enumerate(lines):
                entry = entries[i+1]
                entry.delete(0, tk.END)
                entry.insert(0, line.strip())
    else:
        # If the file doesn't exist, set default values for all parameters
        default_values = ["0"] * NUM_PARAMETERS
        for i, default_value in enumerate(default_values):
            entry = entries[i+1]
            entry.delete(0, tk.END)
            entry.insert(0, default_value)
        # Save the default values to the file
```

```
        save_parameters(default_values)

# Create the main window
window = tk.Tk()
window.title("PARAMETER INPUT FOR THE CURRENT OPTIMISATION: meters, amperes, ohms. You can use the scientific format")

# Create a frame to hold the parameter labels and entry fields
frame = tk.Frame(window, bg="pink")
frame.pack(padx=20, pady=20)

# Create labels and entry fields for each parameter
param_labels = [
    'Tolerance (10^-6 or 10^-8 is recommended) for termination of the iteration process',
    'L0 - length (m) within which the induction was measured',
    'L - optimisation length (m) (choose L > L0)',
    'd0 - wire diameter (m) without isolation or strip width (m) on PCB',
    'delta - thickness (m) of the wire isolation or the half-gap between the strips on PCB',
    'h - strip thickness (m) on PCB',
    'R - internal radius (m) of the shimming coil',
    'Rs - external radius (m) of the shimming coil',
    'w - substrate width (m) including turns; w = d0 + 2*delta if no substrates (only for wires)',
    'rho - conductor resistivity (Ohm*m): 1.68e-8 Cu, 2.65e-8 Al, 1.59e-8 Ag',
    'Current +/- uncertainty (A) caused by the driver performance (estimated)'
]

entries = {}

for i, label_text in enumerate(param_labels):
    label = tk.Label(frame, text=label_text, fg="black", font=("Helvetica", 12), bg="pink", anchor=tk.W)
    label.grid(row=i, column=0, sticky="w", padx=(0, 10), pady=5)

    entry = tk.Entry(frame, font=("Helvetica", 12), bg="pink", width=30)  # Decreased width for value field
    entry.grid(row=i, column=1, padx=(0, 20), pady=5)

    entries[i+1] = entry

# Load the saved parameters as defaults
load_parameters()

# Create a frame as a container for the buttons
button_frame = tk.Frame(window)

# Create the "Run" button
run_button = tk.Button(button_frame, text="Run", command=run_main, font=("Helvetica", 14, "bold"))
run_button.grid(row=0, column=0, padx=10)

# Create the "Stop" button
stop_button = tk.Button(button_frame, text="Stop", command=stop_main, font=("Helvetica", 14, "bold"))
stop_button.grid(row=0, column=1, padx=10)

# Center the button frame within the window
button_frame.pack(pady=10)

# Start the GUI main loop
window.mainloop()
```

NewtonCG.py:

```
#
# The Newton-CG optimization method is used to calculate the current profile when spiral coils are stacked together.
#
from tkinter import simpledialog
import tkinter as tk
import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt
from ResidualB import B0

pi = 3.14159265358979323846264338327795
mu0 = 4.0 * pi * 1.0e-7  # vacuum magnetic permeability

def CurrentOptimisation(tolerance, L0, L, d0, delta, h, R, Rs, w, rho, Inoise):
    #Derived parameters and arrays:
    d = d0 + 2.0 * delta
    R = d * int(R / d) + d / 2.0  # adjusted internal radius
    Rs = d * int(Rs / d) + d / 2.0  # adjusted external radius for the current optimization
    gamma = d / (2.0 * pi)
    theta1 = (2.0 * pi * R) / d - pi
    theta2 = (2.0 * pi * Rs) / d - pi
    gtd1 = 2.0 * gamma * theta1 + d
    gtd2 = 2.0 * gamma * theta2 + d
    turns = int(Rs / d) - int(R / d)  # number of full turns in each spiral for the current optimization

    # Spiral length and resistance
    val1 = (4.0 * gamma**2 + gtd1**2)**0.5
    val2 = (4.0 * gamma**2 + gtd2**2)**0.5
    sp_length = (gtd2/(8.0 * gamma)) * val2 - (gtd1/(8.0 * gamma)) * val1 + \
            (gamma / 2) * np.log((gtd2 + val2) / (gtd1 + val1))
    wire_resistance = rho * sp_length / ((pi / 4.0) * d0**2)
    strip_resistance = rho * sp_length / (d0 * h)

    M = int(L/w)  # number of the spiral coils; must be odd
    M = M + 1 if M % 2 == 0 else M + 2
    zm = np.array(list(map(lambda m: -((M - 1)/2) * w + w * m, range(M))))  # coordinates of the spirals for [-L/2,L/2]
#   print('number of the spiral coils = ', M)

    Q = int(L0/w)  # number of the induction sampling points; must be odd
    Q = Q + 1 if Q % 2 == 0 else Q + 2
    zq = np.array(list(map(lambda q: -((Q - 1)/2) * w + w * q, range(Q))))  # induction sampling points for [-L0/2,L0/2]
    b0 = np.array(list(map(lambda q: B0(zq[q]), range(Q))))  # sampling of the residual induction in Tesla
    averb0 = sum(b0[q] for q in range(Q)) / Q

    plt.plot(zq, b0)
    plt.xlabel("z0, m")
    plt.ylabel("B0, T")
    plt.title("Residual induction")
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    # 2D array of coefficients in Eq. (21) in the report
    Imcoeff = []
    for m in range(M):
        row = []
        for q in range(Q):
            sqrt_term = lambda gtd: (gtd ** 2 + 4.0 * (zq[q] - zm[m]) ** 2) ** 0.5
            value1 = gtd1 + sqrt_term(gtd1)
            value2 = gtd2 + sqrt_term(gtd2)
            value = np.log(value2 / value1) + gtd1 / sqrt_term(gtd1) - gtd2 / sqrt_term(gtd2)
            row.append(value)
```

```
        Imcoeff.append(row)
    Imcoeff = np.array(Imcoeff)

    # Initial values for the current in Eq. (23) in the report
    q0 = int((Q - 1) / 2)  # middle point where z0 = 0
    field = sum(Imcoeff[m][q0] for m in range(M))
    initial_value = -4.0 * pi * gamma * averb0 / (mu0 * field)
    x0 = [initial_value] * M

    def get_initial_current_value(window, initial_value):
        #root = tk.Tk()
        root.withdraw()
        response = simpledialog.askstring("Input",
                                          f"Suggested initial current for the optimization =
{initial_value} A\n"
                                          "If the optimization does not work, propose a new value,"
                                          " choosing it as small as possible.\n"
                                          "You may try zero and then other values in small increments.\n"
                                          "Would you like to change? (y/n)",
                                          parent=window)
        if response.lower() == 'y':
            new_value = simpledialog.askfloat("Input", "Enter your initial current value in A:",
                                              parent=window)
            if new_value is not None:
                return new_value
        return initial_value

    root = tk.Tk()
    def run_main():
        value = get_initial_current_value(root, initial_value)
        I0 = [value] * M
        return I0
    x0 = run_main()

    def on_run_button_click():
        x0 = run_main()

    # 2D array of coefficients in Eq. (21) in the report
    dBz = []
    for m in range(M):
        row = []
        for q in range(Q):
            value = (mu0 / (4.0 * pi * gamma)) * Imcoeff[m][q]
            row.append(value)
        dBz.append(row)
    dBz = np.array(dBz)

    # Hessian matrix in Eq. (22) in the report
    HM = []
    for m in range(M):
        row = []
        for n in range(M):
            value = 2.0 * sum(dBz[m][q] * dBz[n][q] for q in range(Q))
            row.append(value)
        HM.append(row)
    HM = np.array(HM)

    # Hessian as a function but the variable is not used since it is a constant matrix
    def HJ(x):
        return HM

    # Function in Eq. (21) in the report
    # x is the current array used in the external iterations in Eq. (18)
    def Bz(x, q):
        return (mu0 / (4.0 * pi * gamma)) * sum(x[m] * Imcoeff[m][q] for m in range(M))

    # Gradient function in Eq. (21) in the report
```

```
    def GJ(x):
        output = []
        for m in range(M):
            value = sum(2.0 * (Bz(x, q) + b0[q]) * dBz[m][q] for q in range(Q))
            output.append(value)
        return np.array(output)

    # Least square function (15) used for the optimisation
    def fun(x):
        return sum((Bz(x, q) + b0[q])**2 for q in range(Q))

    # Minimisation algorithm:
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#id1
    result = minimize(fun, x0, method = 'Newton-CG', jac = GJ, hess = HJ, tol = tolerance)
    current = np.array(result.x)
    noise_current = (lambda x: x + np.random.uniform(-Inoise, Inoise, size=x.shape))(current)
    wire_current_power = sum(map(lambda I: wire_resistance * I**2, current))
    strip_current_power = sum(map(lambda I: strip_resistance * I ** 2, current))

    with open('design_parameters_for_currents.txt', 'w') as file:
        file.write(f'strip thickness = {h} m\n')
        file.write(f'conductor diameter/width without isolation/gap = {d0} m\n')
        file.write(f'total conductor diameter/width = {d} m\n')
        file.write(f'total substrate width, including the wire diameter = {w} m\n')
        file.write(f'residual induction length = {L0} m\n')
        file.write(f'total optimisation length = {L} m\n')
        file.write(f'adjusted internal radius = {R} m\n')
        file.write(f'adjusted external radius = {Rs} m\n')
        file.write(f'number of spiral turns = {turns}\n')
        file.write(f'each spiral length = {sp_length} m\n')
        file.write(f'wire spiral resistance = {wire_resistance} Ohms\n')
        file.write(f'strip spiral resistance = {strip_resistance} Ohms\n')
        file.write(f'power dissipated in the whole wire stack = {wire_current_power} W\n')
        file.write(f'power dissipated in the whole strip stack = {strip_current_power} W\n')

    error = [lambda q: (Bz(current, q) + b0[q]) for q in range(Q)]
    noise_error = [lambda q: (Bz(noise_current, q) + b0[q]) for q in range(Q)]
    error = np.array([func(q) for q, func in enumerate(error)])
    noise_error = np.array([func(q) for q, func in enumerate(noise_error)])

    def plot_and_save_data(x, y, title, xlabel, ylabel, filename):
        plt.plot(x, y)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.title(title)
        plt.grid(True)
        plt.tight_layout()
        plt.show()

        data = np.column_stack((x, y))
        np.savetxt(filename, data, delimiter=",")

    def plot_and_save_data_with_legend(x1, y1, x2, y2, title, xlabel, ylabel, filename):
        plt.plot(x1, y1, color='blue', label='B0')
        plt.plot(x2, y2, color='green', label='B0 + Bz')
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.title(title)
        plt.legend()
        plt.tight_layout()
        plt.grid(True)
        plt.show()

        data = np.column_stack((x1, y1, y2))
        np.savetxt(filename, data, delimiter=",")

    # Plot 1: Current profile
```

```
    plot_and_save_data(zm, current, "Current profile", "z0, m", "I, A", "current_profile.csv")

    # Plot 2: Error profile
    plot_and_save_data(zq, error, "Error profile", "z0, m", "Error, T", "error_profile.csv")

    # Plot 3: Induction profile before and after optimization
    plot_and_save_data_with_legend(zq, b0, zq, error, "Induction profile before and after optimization",
"z0, m",
                                   "Induction, T", "induction_profile.csv")

    # Plot 4: Induction profile with current uncertainty
    plot_and_save_data_with_legend(zq, b0, zq, noise_error,
                                   "Induction profile before and after optimization:\n with the current
uncertainty",
                                   "z0, m", "Induction, T", "induction_profile_with_uncertainty.csv")
```

ResidualB.py (interpolated distribution of the residual induction):

```
#
# Single-axis residual induction (T, Tesla) as a function of the z0 coordinate.
# You can use a polynomial or any other fitting function of a single variable to describe the residual field.
#

# A residual field provided by the mmWave lab at the University of St. Andrews, Scotland.
def B0(z0):
    value = -5.859e-6 + 4.766114e-3 * z0 - 0.486506371 * z0**2 - 1.4609783504e1 * z0**3 + 4.2600403748e2 *
z0**4 +\
            3.04437e4 * z0**5 - 8.75637e5 * z0**6 - 7.65903e6 * z0**7 + 1.46997e8 * z0**8
    return value
```

Principle of PWM DC source

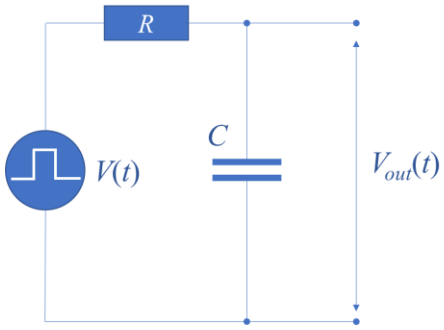
The PWM signal is a periodic extension of the following step function defined over the interval T, which serves as the basis for the periodic extension period:

$$V(t) = \begin{cases} A, 0 \le t \le \tau \\ 0, \tau < t < T \end{cases}$$

where A is a positive constant (5V for Arduino or 3.3V for Raspberry Pi). The ratio τ/T (%) is called the duty cycle. The Fourier spectrum of this function is represented by the following series:

$$V(t) = \frac{A\tau}{T} + \sum_{k=1}^{\infty} \left[\frac{A}{k\pi} \sin\left(\frac{2\pi k\tau}{T}\right) \cos\left(\frac{2\pi k}{T} t\right) + \frac{A}{k\pi} \left(1 - \cos\left(\frac{2\pi k\tau}{T}\right)\right) \sin\left(\frac{2\pi k}{T} t\right) \right]$$

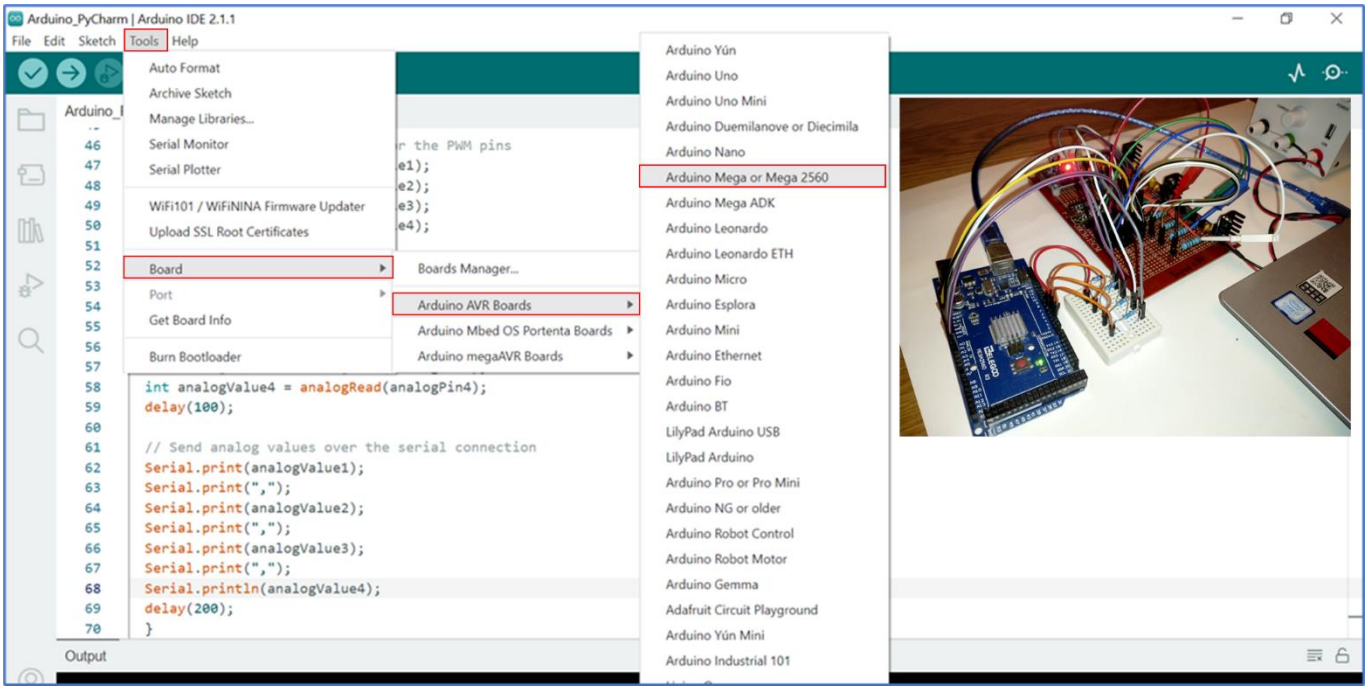
By employing an RC filter (see the figure below) with sufficiently high capacitance and resistance values, the elimination of all harmonics except the zeroth $A\tau/T$ is achieved, leading to the desired DC voltage level. This level is determined by the duty cycle. The load resistance must not be excessively small, as this would hasten the discharge of the capacitor, leading to substantial fluctuations (ripple) in the output voltage $V_{out}(t)$. To mitigate the load's influence on $V_{out}(t)$, consider utilizing a non-inverting voltage follower (buffer) implemented with an operational amplifier (op amp) with short-connected negative feedback, characterized by a megaohm-level input resistance.

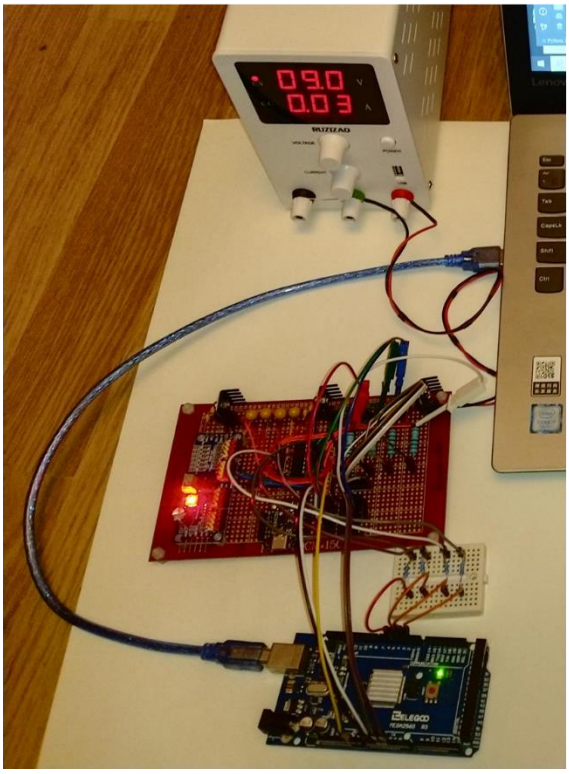


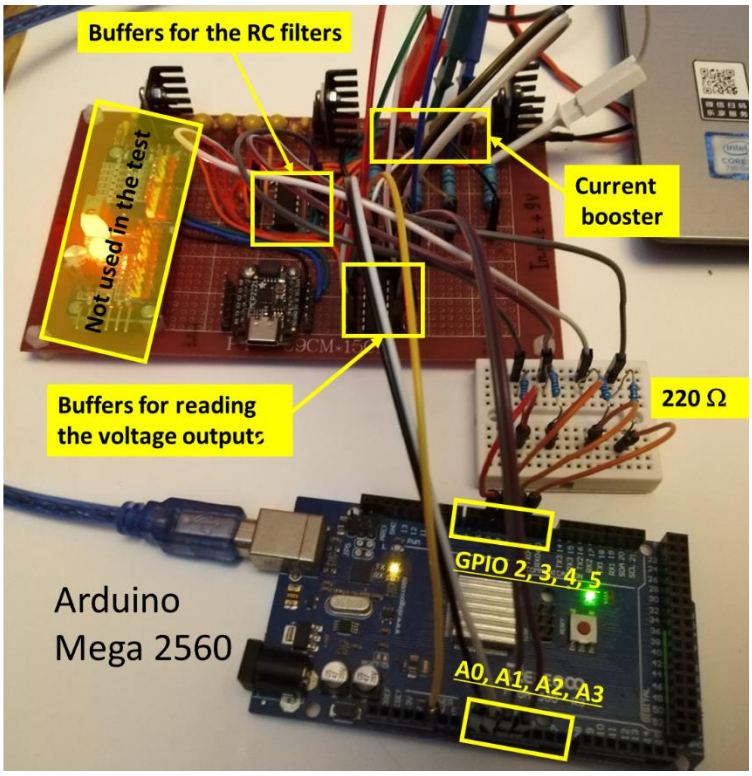
Arduino's sketch and Python program for testing a PWM driver

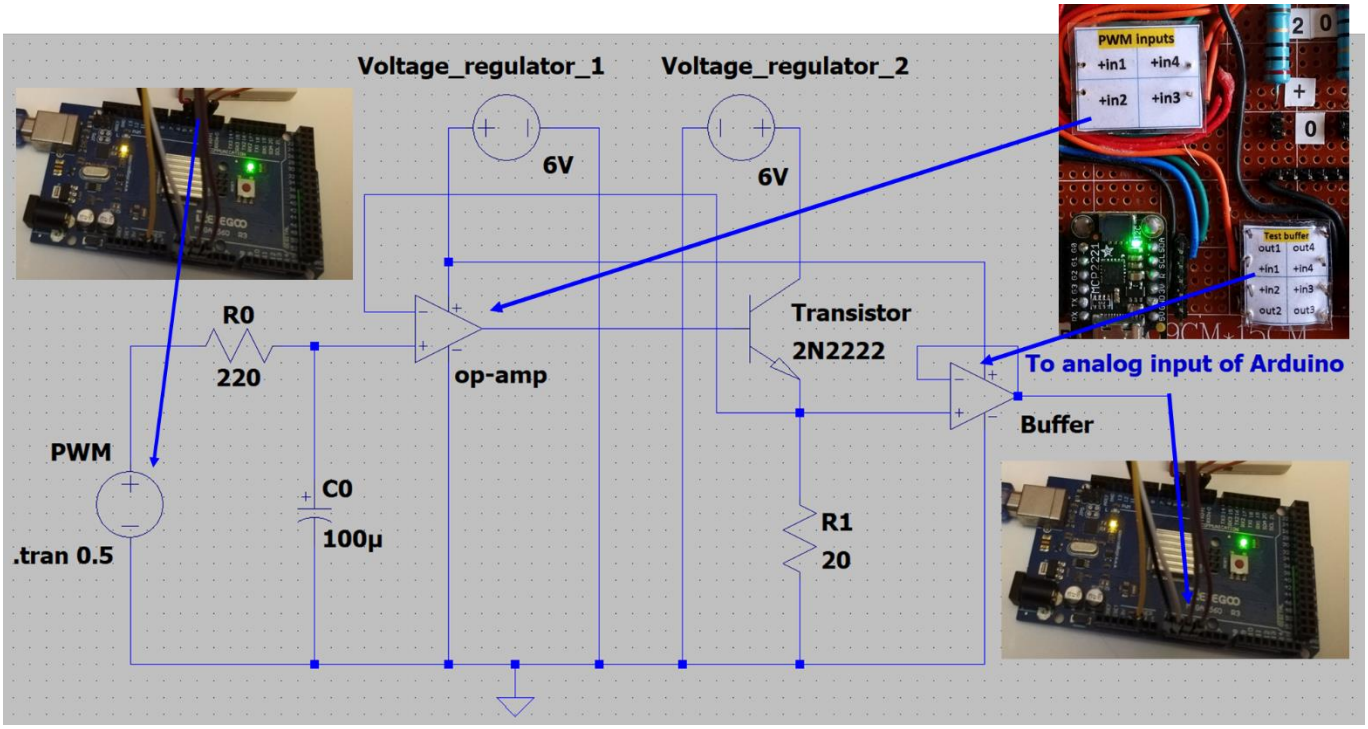
Project folder: "Testing_PWM_driver_Arduino_Python". The Arduino sketch and Python program are employed to assess the functionality of four PWM channels in a driver. These channels are configured with consistent duty cycle values spanning the entire range, totalling 256 discrete steps. Adequate intervals (6τ) are deliberately chosen between these steps to ensure thorough charging of capacitors within the RC filters. The Python program governs the assignment of cycle values and acquires the driver output voltage readings from the Arduino's ADC. This combined framework facilitates driver testing under both amplified and non-amplified current conditions. In either scenario, the Arduino fulfils the dual roles of generating PWM pulses and digitally capturing signals. The PWM pulses are taken from the Arduino's GPIO 2, 3, 4, 5. The driver output voltages are measured from the Arduino's analog inputs A0, A1, A2, A3.

The program has undergone testing on the Arduino Mega 2560; however, with suitable adjustments to the IDE settings and subsequent microcontroller upload, it is adaptable for deployment on any Arduino board.

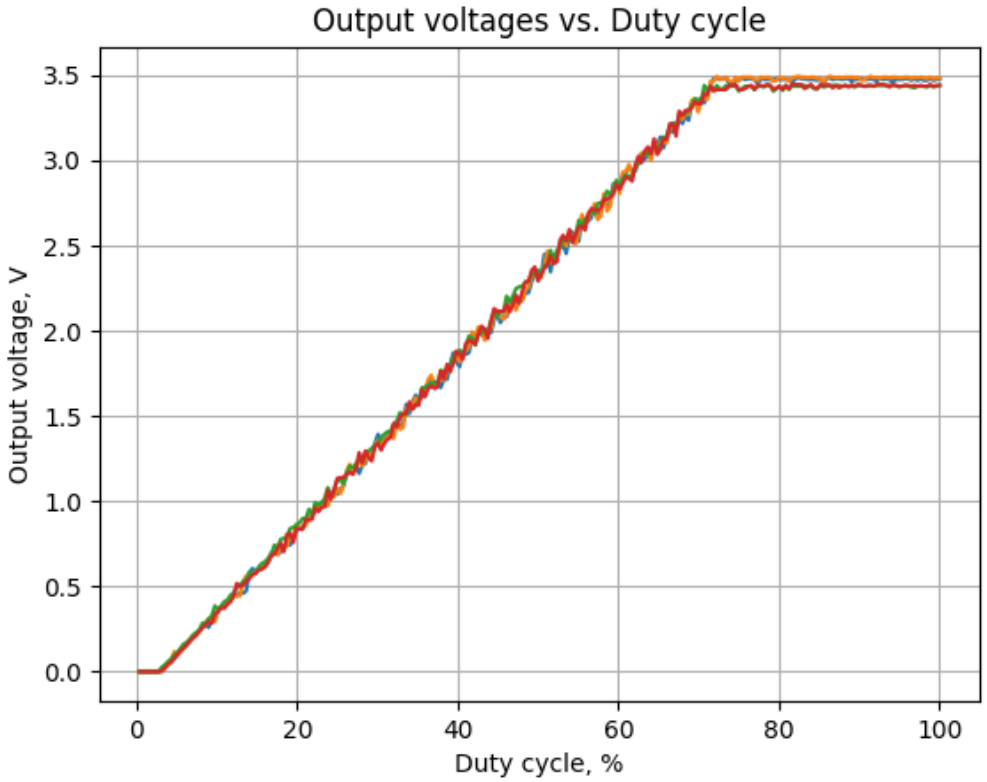








Testing a driver equipped with the current boosters is conducted using identical principles and the same programs (Arduino and Python) as those utilized when testing without the boosters.



The driver outputs four channels with the current boosters. PWM frequency 1500 Hz.

Arduino:

```
#include <Arduino.h>

// Program for Arduino Mega 2560 R3
// It can be easily adapted to other Arduino models

const int pwmPin1 = 2;      // PWM pin 1
const int pwmPin2 = 3;      // PWM pin 2
const int pwmPin3 = 4;      // PWM pin 3
const int pwmPin4 = 5;      // PWM pin 4

const int analogPin1 = A0;  // Analog pin 1
const int analogPin2 = A1;  // Analog pin 2
const int analogPin3 = A2;  // Analog pin 3
const int analogPin4 = A3;  // Analog pin 4
const int pwmFrequency = 1500;  // PWM frequency in Hz

int dutyCycle1 = 0;         // Duty cycle for PWM pin 1
int dutyCycle2 = 0;         // Duty cycle for PWM pin 2
int dutyCycle3 = 0;         // Duty cycle for PWM pin 3
int dutyCycle4 = 0;         // Duty cycle for PWM pin 4

void setup() {
  // Set the PWM frequency for pins 2 to 13 (Timer1)
  TCCR1B = (TCCR1B & 0xF8) | 0x01;  // Set Timer1 prescaler to 1 (no prescaling)

  // Set the PWM frequency for pins 44 to 46 (Timer5)
  TCCR5B = (TCCR5B & 0xF8) | 0x01;  // Set Timer5 prescaler to 1 (no prescaling)

  pinMode(pwmPin1, OUTPUT);
  pinMode(pwmPin2, OUTPUT);
```

```
  pinMode(pwmPin3, OUTPUT);
  pinMode(pwmPin4, OUTPUT);

  // Initialize serial communication
  Serial.begin(9600);
}

void loop() {
  if (Serial.available() > 0) {
    // Read new duty cycle values from the serial port
    dutyCycle1 = Serial.parseInt();
    dutyCycle2 = Serial.parseInt();
    dutyCycle3 = Serial.parseInt();
    dutyCycle4 = Serial.parseInt();

    // Set the new duty cycles for the PWM pins
    analogWrite(pwmPin1, dutyCycle1);
    analogWrite(pwmPin2, dutyCycle2);
    analogWrite(pwmPin3, dutyCycle3);
    analogWrite(pwmPin4, dutyCycle4);
    delay(100);
  }

  // Read analog values from the analog pins
  int analogValue1 = analogRead(analogPin1);
  int analogValue2 = analogRead(analogPin2);
  int analogValue3 = analogRead(analogPin3);
  int analogValue4 = analogRead(analogPin4);
  delay(100);

  // Send analog values over the serial connection
  Serial.print(analogValue1);
  Serial.print(",");
  Serial.print(analogValue2);
  Serial.print(",");
  Serial.print(analogValue3);
  Serial.print(",");
  Serial.println(analogValue4);
  delay(200);
  }

```

main.py:

```
#
# This Python code is used to test the PWM driver and draw its voltage characteristics.
# Use this Python program to control four PWM outputs on Arduino and measure its four analog inputs.
# Plotting graphs Vout(PWM) vs. duty cycle.
# Four PWM outputs on Arduino: duty cycle sweep from 0 to 100%.
#

import serial
import time
import numpy as np
import matplotlib.pyplot as plt

# This time delay (s) must be much larger (x5 or more) than the RC characteristic time of the RC filter
time_delay = 0.2

# Establish serial connection with Arduino
ser = serial.Serial('COM3', 9600)

# Wait for the Arduino to reset
time.sleep(1)

```

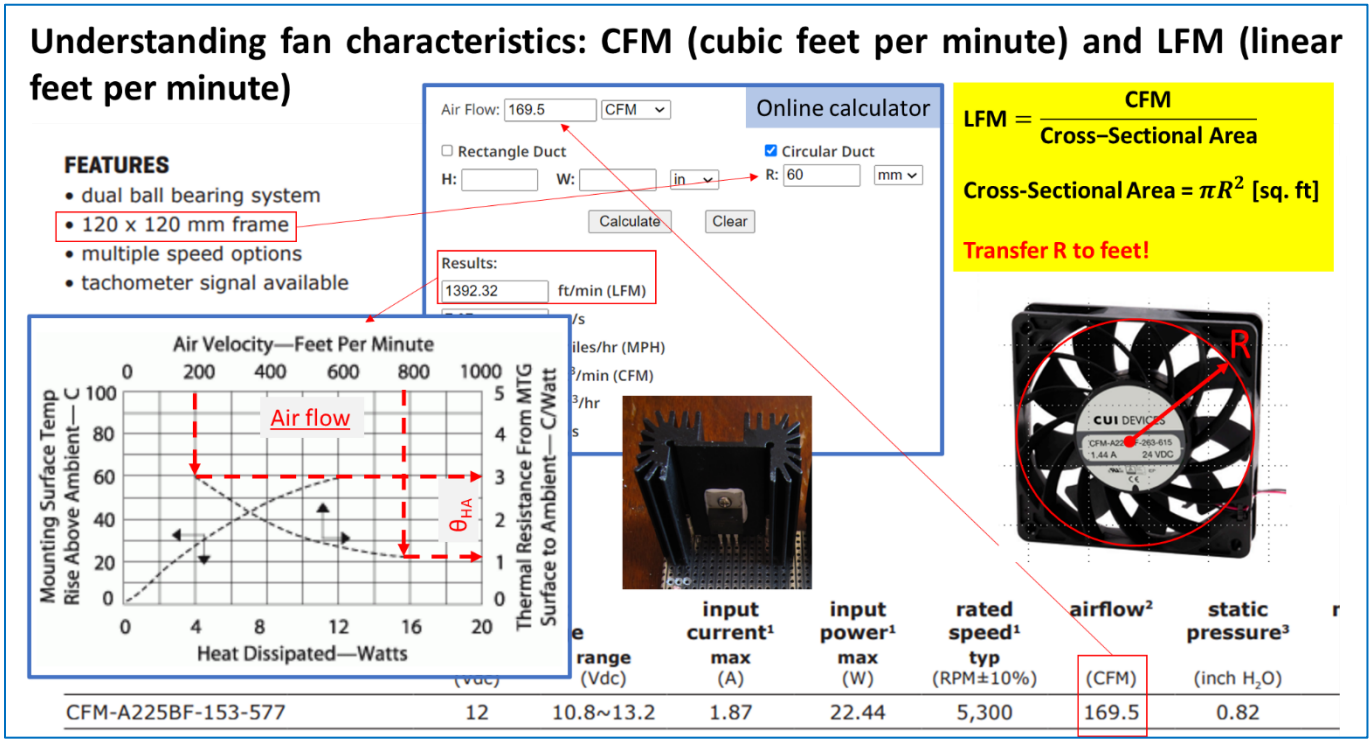
```
# Define constant duty cycle values
duty_cycle1 = 0  # PWM pin 1
duty_cycle2 = 0  # PWM pin 2
duty_cycle3 = 0  # PWM pin 3
duty_cycle4 = 0  # PWM pin 4

# Set initial duty cycles on Arduino
ser.write(f"{duty_cycle1} {duty_cycle2} {duty_cycle3} {duty_cycle4} \n".encode())
time.sleep(1)

# ADC properties
ADC_resolution = 1023  # ADC resolution (10-bit)
reference_voltage = 5.0  # Reference voltage (5V)

# Main loop
duty_cycle = []  # 8-bit resolution
volt1 = []
volt2 = []
volt3 = []
volt4 = []
for i in range(1, 257):
    # Read analog values from Arduino
    data = ser.readline().decode().strip().split(",")
    time.sleep(1)

    if len(data) == 4:
        try:
            analog_value1 = int(data[0])
            analog_value2 = int(data[1])
            analog_value3 = int(data[2])
            analog_value4 = int(data[3])

            # Convert analog readings to voltage
            voltage1 = (analog_value1 / ADC_resolution) * reference_voltage
            voltage2 = (analog_value2 / ADC_resolution) * reference_voltage
            voltage3 = (analog_value3 / ADC_resolution) * reference_voltage
            voltage4 = (analog_value4 / ADC_resolution) * reference_voltage
            volt1.append(voltage1)
            volt2.append(voltage2)
            volt3.append(voltage3)
            volt4.append(voltage4)

            # Print voltage values
            print(f"Voltage 1: {voltage1}V")
            print(f"Voltage 2: {voltage2}V")
            print(f"Voltage 3: {voltage3}V")
            print(f"Voltage 4: {voltage4}V")

            # Control duty cycles
            value = (i * 100.0)/256
            duty_cycle.append(value)
            new_duty_cycle1 = i  # Calculate the desired duty cycle for PWM pin 1
            new_duty_cycle2 = i  # Calculate the desired duty cycle for PWM pin 2
            new_duty_cycle3 = i  # Calculate the desired duty cycle for PWM pin 3
            new_duty_cycle4 = i  # Calculate the desired duty cycle for PWM pin 4
            print(f"Duty cycle: {value}%")
            print('')

            # Send new duty cycles to Arduino
            ser.write(f"{new_duty_cycle1} {new_duty_cycle2} {new_duty_cycle3} {new_duty_cycle4}\n".encode())
            time.sleep(time_delay)

        except ValueError:
            print("Invalid data received from Arduino")

    else:
        print("Invalid data format received from Arduino")
```

```
 89.
 90. # Set initial duty cycles on Arduino
 91. duty_cycle1 = 0  # PWM pin 1
 92. duty_cycle2 = 0  # PWM pin 2
 93. duty_cycle3 = 0  # PWM pin 3
 94. duty_cycle4 = 0  # PWM pin 4
 95.
 96. ser.write(f"{duty_cycle1} {duty_cycle2} {duty_cycle3} {duty_cycle4} \n".encode())
 97. time.sleep(1)
 98.
 99. # Close the serial connection
100. ser.close()
101.
102. duty_cycle = np.array(duty_cycle)
103. volt1 = np.array(volt1)
104. volt2 = np.array(volt2)
105. volt3 = np.array(volt3)
106. volt4 = np.array(volt4)
107.
108. # Plotting the graph
109. plt.plot(duty_cycle, volt1)
110. plt.plot(duty_cycle, volt2)
111. plt.plot(duty_cycle, volt3)
112. plt.plot(duty_cycle, volt4)
113. plt.xlabel("Duty cycle, %")
114. plt.ylabel("Output voltage, V")
115. plt.title("Output voltages vs. Duty cycle")
116. plt.grid(True)
117.
118. # Displaying the graph
119. plt.show()
120.
```

Heat sinks with forced cooling

The durability of semiconductor devices is directly correlated with the operating temperature of their junctions – the lower the temperature, the greater the durability of the circuit. However, implementing technical measures to decrease the operating temperature invariably results in an increased cost for the solution.

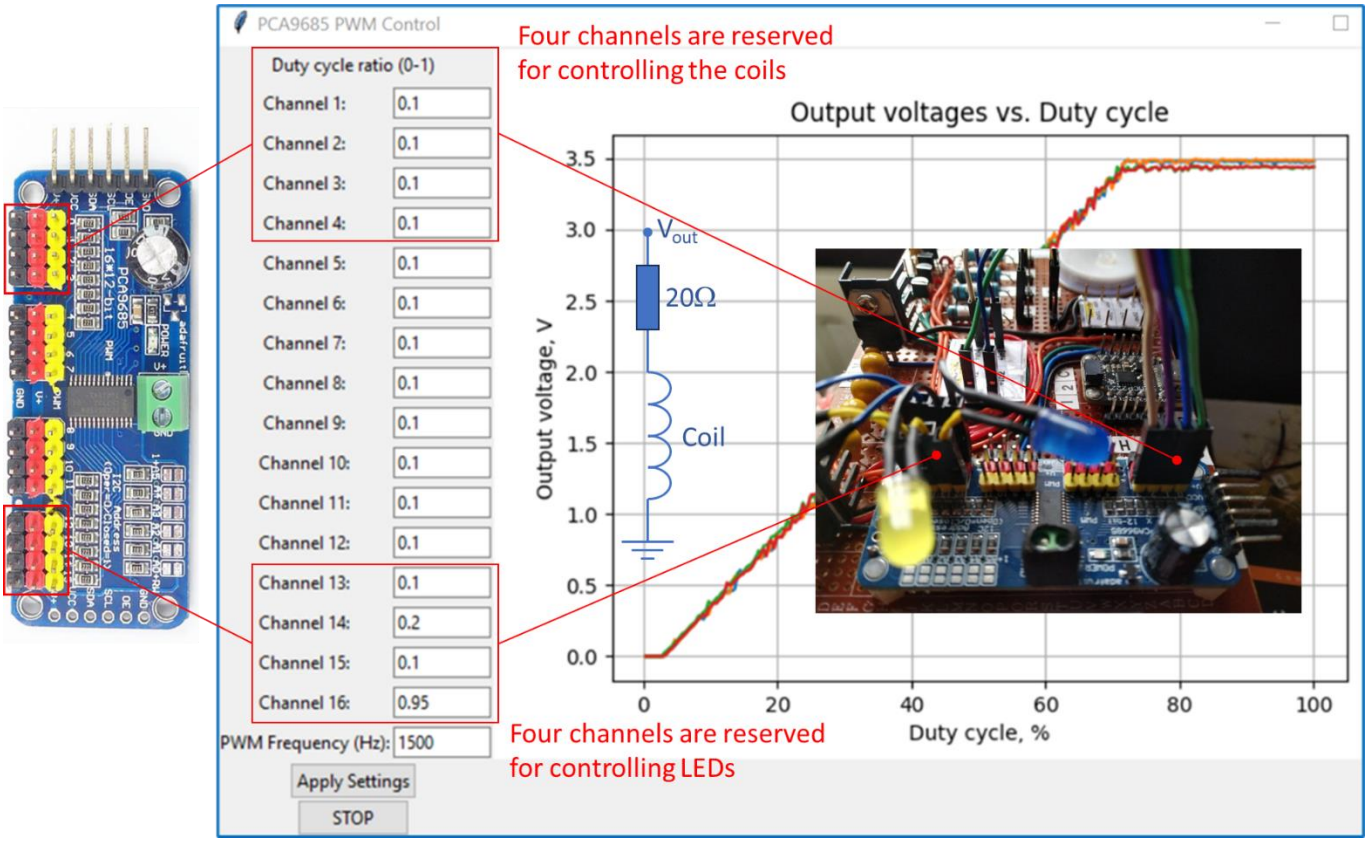
Utilizing a combination of a heat sink and forced airflow substantially reduces the heat sink-to-ambient thermal resistance θ_{HA} (℃/W) in Eq. (28). This reduction in thermal resistance addresses the primary factor contributing to heat buildup. The airflow intensity from the fan can be quantified using two units: CFM (cubic feet per minute) and LFM (linear feet per minute). These units can be converted into each other, as explained in the Figure below, where we present the characteristics of the High Power Extruded 6390 heat sink as an example. You can also find online calculators for conversion. According to the graph, when the air flow rate is 1392.32 LFM (already outside the scope of the abscissa range), θ_{HA} will be less than 1 ℃/W, which is significantly lower than the value for free convection (~5 ℃/W when extrapolated to zero LFM on the graph).



Control program and GUI for the electronic driver with the current boosters

Project folder: "PWM_driver_Current_booster_Blinka_Shimming_Coils_Python". The Python program is responsible for managing the functionality of the electronic driver depicted in Fig. 21. The duty cycle values for all 16 channels are stored in the "duty_cycle.log" file. When you relaunch the program, GUI will display these values as the default settings. You can modify these values by inputting new ones and then clicking the "Apply Settings" button, at which point the updated values will be saved once again in the log file. The default PWM frequency is set at 1500 Hz. The actual frequency on the microchip will be 1525.88 Hz, which has been recorded in the log file. The microchip PCA9685 does permit the use of several frequency values lower than this, but it is advisable to maintain the default frequency setting. To terminate the program and halt the GUI's loop, simply click the "STOP" button. Please ensure that you do not delete the file "VoltageOutput.png" from the project folder as it is utilized by the GUI when it is running.

The functional blocks within the driver (refer to Fig. 21) are intentionally not permanently interconnected. This design choice offers greater flexibility, allowing for individual learning of each block. The circuit can be configured for various types of measurements, and users are encouraged to experiment with microchips and devise their own measurement scenarios.



main.py:

```
#
# The program with GUI to control the electronic driver for the shimming coils (four channels).
# PWM driver with the current booster.
# Connect Blinka USB-I2C adapter to PC.
#
# Yujie Zhao, University of St. Andrews, Scotland, 30.07.2023
#

import os
import busio
import board
import adafruit_pca9685
import tkinter as tk
from tkinter import ttk
import logging

# Set BLINKA_MCP2221 environment variable to enable MCP2221 support
os.environ['BLINKA_MCP2221'] = '1'

# Create the I2C bus interface.
i2c_bus = busio.I2C(board.SCL, board.SDA)

# Specify the PCA9685 address (0x40) when creating the instance.
pca = adafruit_pca9685.PCA9685(i2c_bus, address=0x40)

# Set the default frequency to 1500 Hz
pca.frequency = 1500

# Function to update the duty cycle for a channel based on the user input
def update_duty_cycle(channel):
    try:
        duty_cycle = float(duty_entries[channel - 1].get())
        pca.channels[channel - 1].duty_cycle = int(duty_cycle * 0xFFFF)  # Convert to 16-bit value

        # Log the duty cycle change
        logging.info(f"Channel {channel} duty cycle set to {duty_cycle}")

        # Write the new value to the log file
        with open('duty_cycle.log', 'a') as log_file:
            log_file.write(f"Channel {channel} duty cycle set to {duty_cycle}\n")

    except ValueError:
        pass  # If the user enters an invalid value, ignore it

# Function to update the PWM frequency based on the user input
def update_pwm_frequency():
    try:
        frequency = float(freq_entry.get())

        # Write the PWM frequency change to the log file
        logging.info(f"PWM frequency set to {frequency:.2f} Hz")

        # Set the new PWM frequency directly to the PCA9685 instance
        pca.frequency = frequency

        # Write the new value to the log file with up to two decimal places
        with open('duty_cycle.log', 'a') as log_file:
            log_file.write(f"PWM frequency set to {frequency:.2f} Hz\n")

    except ValueError:
        pass  # If the user enters an invalid value, ignore it

# Function to apply the updated duty cycles and PWM frequency to PCA9685
def apply_settings():
    update_pwm_frequency()
```

```
    for channel in range(1, 17):
        update_duty_cycle(channel)

    # Fully rewrite the log file with the new values
    with open('duty_cycle.log', 'w') as log_file:
        log_file.write(f"PWM frequency set to {pca.frequency:.2f} Hz\n")
        for channel in range(1, 17):
            duty_cycle = pca.channels[channel - 1].duty_cycle / 0xFFFF
            log_file.write(f"Channel {channel} duty cycle set to {duty_cycle:.2f}\n")

    # Read default values from the log file (if it exists)
    read_default_values()

# Function to stop the program
def stop_program():
    root.quit()

# Function to read default values from the log file
def read_default_values():
    if os.path.exists('duty_cycle.log'):
        with open('duty_cycle.log', 'r') as log_file:
            lines = log_file.readlines()
            for line in lines:
                if "duty cycle" in line:
                    channel = int(line.split()[1])
                    duty_cycle = float(line.split()[-1])
                    duty_entries[channel - 1].delete(0, tk.END)
                    duty_entries[channel - 1].insert(0, str(duty_cycle))
                elif "PWM frequency" in line and not freq_entry.get():
                    frequency = float(line.split()[-2])
                    freq_entry.delete(0, tk.END)
                    freq_entry.insert(0, f"{frequency:.2f}")

# GUI setup
root = tk.Tk()
root.title("PCA9685 PWM Control")

# Label for the duty cycle explanation
ttk.Label(root, text="Duty cycle ratio (0-1)").grid(row=0, column=0, columnspan=2)

# Labels and Entry widgets for duty cycles
duty_entries = []
for channel in range(1, 17):
    ttk.Label(root, text=f"Channel {channel}:").grid(row=channel, column=0)
    entry = ttk.Entry(root, width=10)
    entry.insert(0, "0.5")  # Default value is 50%
    entry.grid(row=channel, column=1)
    duty_entries.append(entry)

# Label and Entry widget for PWM frequency
ttk.Label(root, text="PWM Frequency (Hz):").grid(row=17, column=0)
freq_entry = ttk.Entry(root, width=10)
freq_entry.insert(0, "1500")  # Default frequency is 1500 Hz
freq_entry.grid(row=17, column=1)

# Load the image and create a label to display it
image_path = "VoltageOutput.png"
if os.path.exists(image_path):
    voltage_output_image = tk.PhotoImage(file=image_path)
    image_label = tk.Label(root, image=voltage_output_image)
    image_label.grid(row=0, column=2, rowspan=18)  # Adjust the row and column as needed

# Button to apply duty cycles and PWM frequency
apply_button = ttk.Button(root, text="Apply Settings", command=apply_settings)
apply_button.grid(row=18, column=0, columnspan=2)

# Button to stop the program
```

```
stop_button = ttk.Button(root, text="STOP", command=stop_program)
stop_button.grid(row=19, column=0, columnspan=2)

# Read default values from the log file at the beginning
read_default_values()

# Run the GUI main loop
root.mainloop()

```

Symbolic calculus using SymPy library

Project folder: "dBz_derivative". SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. We used SymPy to calculate the partial derivatives in Eqs. (33), (34).

main.py:

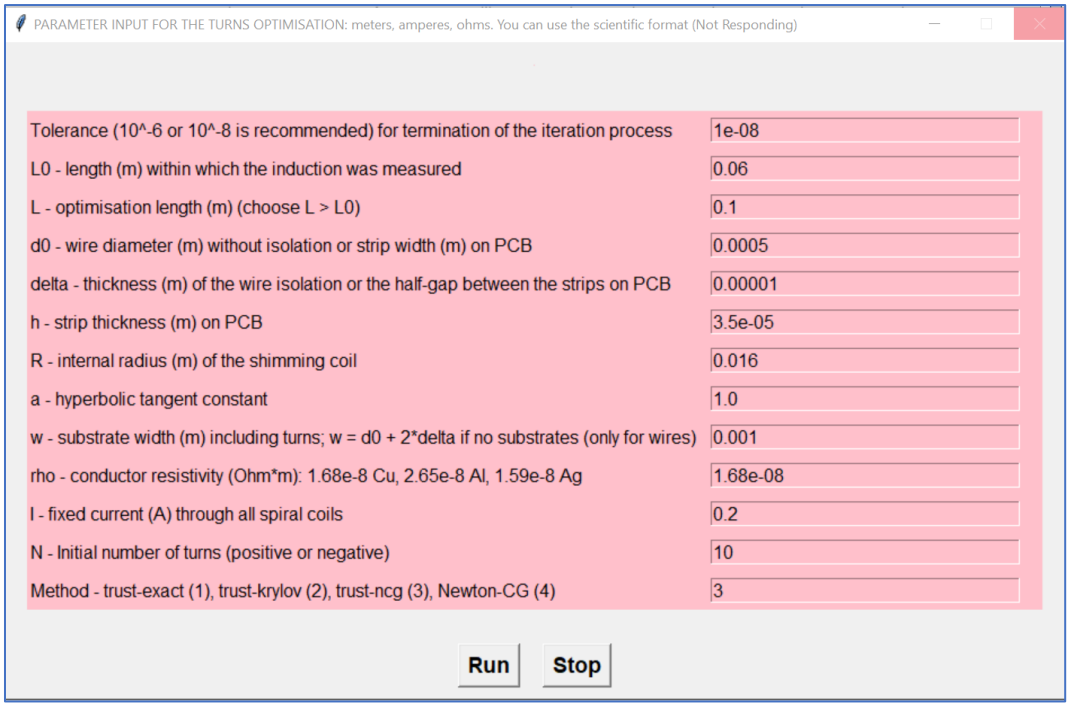
```
#
# This code is used to symbolically calculate the derivatives in Eqs. (33), (34)
#

import sympy as sp
import textwrap

# Define the variables in your expression
x, d, gamma, theta1, theta2, gtd1, gtd2, a, zm, zq, pi, mu0, I = sp.symbols('x, d, gamma, theta1, theta2, gtd1, gtd2,'
                                                                            ' a, zm, zq, pi, mu0, I')

def bz(x):
    theta2 = theta1 + x * sp.tanh(a * x)
    gtd2 = 2.0 * gamma * theta2 + d
    value1 = gtd1 + (gtd1 ** 2 + 4.0 * (zq - zm) ** 2) ** 0.5
    value2 = gtd2 + (gtd2 ** 2 + 4.0 * (zq - zm) ** 2) ** 0.5
    value = (sp.log(value2 / value1) + gtd1 / ((gtd1 ** 2 + 4.0 * (zq - zm) ** 2) ** 0.5)
             - gtd2 / ((gtd2 ** 2 + 4.0 * (zq - zm) ** 2) ** 0.5))
    bz_value = sp.tanh(a * x) * value
    return mu0 * I * bz_value / (4.0 * pi * gamma)

expression = bz(x)

# Calculate the partial derivative with respect to x
partial_derivative = sp.diff(expression, x)

# Simplify the result
simplified_derivative = sp.simplify(partial_derivative)

# Convert the simplified result to a string
result_str = str(simplified_derivative)

# Define the maximum line width
max_line_width = 80  # Adjust this value to fit your window width

# Wrap the lines to fit within the specified width
wrapped_lines = textwrap.wrap(result_str, width=max_line_width)

# Print the wrapped lines
for line in wrapped_lines:
    print(line)
```

Below, you can see the calculation result. Afterward, all variables and array components were replaced with the ones used in the “Optimisation.py” module in the “Turns_Optimisation_Python” project folder.

```
0.25*I*mu0*(a*((0.5*d + gamma*(theta1 + x*tanh(a*x)))**2 + (zm - zq)**2)**2.5*(d
+ 2.0*gamma*(theta1 + x*tanh(a*x)) + 2.0*((0.5*d + gamma*(theta1 +
x*tanh(a*x)))**2 + (zm - zq)**2)**0.5)*(0.5*gtd1*((0.5*d + gamma*(theta1 +
x*tanh(a*x)))**2 + (zm - zq)**2)**0.5 - (0.5*d + 1.0*gamma*(theta1 +
x*tanh(a*x)))*(0.25*gtd1**2 + (zm - zq)**2)**0.5 + (0.25*gtd1**2 + (zm -
zq)**2)**0.5*((0.5*d + gamma*(theta1 + x*tanh(a*x)))**2 + (zm -
zq)**2)**0.5*log((d + 2.0*gamma*(theta1 + x*tanh(a*x)) + 2.0*((0.5*d +
gamma*(theta1 + x*tanh(a*x)))**2 + (zm - zq)**2)**0.5)/(gtd1 + 2.0*(0.25*gtd1**2
+ (zm - zq)**2)**0.5))))/cosh(a*x)**2 + gamma*(0.25*gtd1**2 + (zm -
zq)**2)**0.5*(a*x/cosh(a*x)**2 + tanh(a*x))*((0.5*d + gamma*(theta1 +
x*tanh(a*x)))**2 + (zm - zq)**2)**0.5*(0.5*(0.5*d + gamma*(theta1 +
x*tanh(a*x)))*(d + 2.0*gamma*(theta1 + x*tanh(a*x)))*((0.5*d + gamma*(theta1 +
x*tanh(a*x)))**2 + (zm - zq)**2)**1.0*(d + 2.0*gamma*(theta1 + x*tanh(a*x)) +
2.0*((0.5*d + gamma*(theta1 + x*tanh(a*x)))**2 + (zm - zq)**2)**0.5) +
2.0*((0.5*d + gamma*(theta1 + x*tanh(a*x)))**2 + (zm - zq)**2)**2.0*(0.5*d +
gamma*(theta1 + x*tanh(a*x)) + ((0.5*d + gamma*(theta1 + x*tanh(a*x)))**2 + (zm
- zq)**2)**0.5) - 1.0*((0.5*d + gamma*(theta1 + x*tanh(a*x)))**2 + (zm -
zq)**2)**2.0*(d + 2.0*gamma*(theta1 + x*tanh(a*x)) + 2.0*((0.5*d + gamma*(theta1
+ x*tanh(a*x)))**2 + (zm - zq)**2)**0.5))*tanh(a*x))/(gamma*pi*(0.25*gtd1**2 +
(zm - zq)**2)**0.5*((0.5*d + gamma*(theta1 + x*tanh(a*x)))**2 + (zm -
zq)**2)**3.0*(d + 2.0*gamma*(theta1 + x*tanh(a*x)) + 2.0*((0.5*d + gamma*(theta1
+ x*tanh(a*x)))**2 + (zm - zq)**2)**0.5))
```

Turns optimisation algorithm for the shimming coil

Project folder: "Turns_Optimisation_Python". The program features a graphical user interface (GUI) that can be launched by executing the "main.py" file. Other modules in the program project: "Optimisation.py" (turns optimization algorithm) and "ResidualB.py" (distribution of the residual induction). Within the folder, there is a "parameters.txt" log file that should not be removed. This file retains the default parameters for use in subsequent program launches. The file "design_parameters_for_turns.txt" holds the coil's design parameters, which are obtained through user interaction and optimization. The calculated turns profiles and corresponding fields are stored in CSV files with self-explanatory names. GUI (main.py) is similar to that used for the current optimization but with some notable changes. The outer diameter of each coil is now a variable subject to optimization. Additionally, three new parameters have been introduced: the hyperbolic tangent constant a, the initial number of turns N (which can be positive or negative), and the utilization of four gradient unconstrained methods.



As with the current optimization, there is not just one, but an entire range ("valley") of acceptable solutions. The specific solution that the optimization converges to depends on the choice of wire diameter or strip width, isolation thickness, initial values for turns, current strength, hyperbolic constant, and optimization method (now there are four to choose from). If the hyperbolic constant is selected too large, the program may report an overflow (red messages in console) but will continue to execute. The Newton-CG method performed less effectively than other gradient methods, which is why we have ranked it as number four in the GUI window.

main.py:

```
#
# The main module for GUI that calls the optimisation algorithms
# Yujie Zhao, University of St. Andrews, Scotland, 30th July 2023.
#

import os
import tkinter as tk
from Optimisation import *

PARAMETERS_FILE = "parameters.txt"
NUM_PARAMETERS = 13

def stop_main():
    quit()

def run_main():
    # This function will be called when the "Run" button is clicked
    # You can access the parameter values here and call your own function using the parameters

    # Retrieve the parameter values from the input fields
    params = [float(entries[i].get()) for i in range(1, NUM_PARAMETERS - 1)]  # first part of the list
    params.extend([int(entries[i].get()) for i in range(NUM_PARAMETERS - 1, NUM_PARAMETERS + 1)])  # second
part

    # Save the parameter values to a file
    save_parameters(params)

    # params[0] tolerance: tolerance for termination of the iteration process.
    # params[1] L0: length (m) within which the induction was measured
    # params[2] L: length (m) of the optimisation (we suggest L > L0)
    # params[3] d0: wire diameter (m) without isolation or strip width
    # params[4] delta: thickness of the wire isolation or the half-gap between the strips (m) on PCB
    # params[5] h: strip thickness on PCB (m)
    # Radius R will be slightly adjusted: R = d * int(R/d) + d/2
    # params[6] R: internal radius (m) of the shimming coil
    # params[7] a: hyperbolic tangent constant
    # params[8] w: substrate width (m); w = d0 + 2*delta if no substrates
    # params[9] rho: conductor resistivity (Ohm*m)
    # params[10] I: fixed current (A) through all spiral coils
    # params[11] N: Initial number of turns (positive or negative)
    # params[12] Method: trust-exact (1), trust-krylov (2), trust-ncg (3), Newton-CG (4)

    TurnsOptimisation(params[0], params[1], params[2], params[3], params[4], params[5], params[6], params[7],
                      params[8], params[9], params[10], params[11], params[12])

def save_parameters(params):
    # Save the parameter values to a file
    with open(PARAMETERS_FILE, "w") as file:
        for param in params:
            file.write(f"{param}\n")

def load_parameters():
    # Load the parameter values from the file, if it exists
    if os.path.exists(PARAMETERS_FILE):
        with open(PARAMETERS_FILE, "r") as file:
            lines = file.readlines()
            for i, line in enumerate(lines):
                entry = entries[i+1]
                entry.delete(0, tk.END)
                entry.insert(0, line.strip())
    else:
        # If the file doesn't exist, set default values for all parameters
        default_values = ["0"] * NUM_PARAMETERS
        for i, default_value in enumerate(default_values):
            entry = entries[i+1]
```

```
            entry.delete(0, tk.END)
            entry.insert(0, default_value)
        # Save the default values to the file
        save_parameters(default_values)

# Create the main window
window = tk.Tk()
window.title("PARAMETER INPUT FOR THE TURNS OPTIMISATION: meters, amperes, ohms. You can use the scientific format")

# Create a frame to hold the parameter labels and entry fields
frame = tk.Frame(window, bg="pink")
frame.pack(padx=20, pady=20)

# Create labels and entry fields for each parameter
param_labels = [
    'Tolerance (10^-6 or 10^-8 is recommended) for termination of the iteration process',
    'L0 - length (m) within which the induction was measured',
    'L - optimisation length (m) (choose L > L0)',
    'd0 - wire diameter (m) without isolation or strip width (m) on PCB',
    'delta - thickness (m) of the wire isolation or the half-gap between the strips on PCB',
    'h - strip thickness (m) on PCB',
    'R - internal radius (m) of the shimming coil',
    'a - hyperbolic tangent constant',
    'w - substrate width (m) including turns; w = d0 + 2*delta if no substrates (only for wires)',
    'rho - conductor resistivity (Ohm*m): 1.68e-8 Cu, 2.65e-8 Al, 1.59e-8 Ag',
    'I - fixed current (A) through all spiral coils',
    'N - Initial number of turns (positive or negative)',
    'Method - trust-exact (1), trust-krylov (2), trust-ncg (3), Newton-CG (4)'
]

frame = tk.Frame(window, bg="pink")
frame.pack(padx=20, pady=20)

entries = {}

for i, label_text in enumerate(param_labels):
    label = tk.Label(frame, text=label_text, fg="black", font=("Helvetica", 12), bg="pink", anchor=tk.W)
    label.grid(row=i, column=0, sticky="w", padx=(0, 10), pady=5)

    entry = tk.Entry(frame, font=("Helvetica", 12), bg="pink", width=30)  # Decreased width for value field
    entry.grid(row=i, column=1, padx=(0, 20), pady=5)

    entries[i + 1] = entry

# Load the saved parameters as defaults
load_parameters()

# Create a frame as a container for the buttons
button_frame = tk.Frame(window)

# Create the "Run" button
run_button = tk.Button(button_frame, text="Run", command=run_main, font=("Helvetica", 14, "bold"))
run_button.grid(row=0, column=0, padx=10)

# Create the "Stop" button
stop_button = tk.Button(button_frame, text="Stop", command=stop_main, font=("Helvetica", 14, "bold"))
stop_button.grid(row=0, column=1, padx=10)

# Center the button frame within the window
button_frame.pack(pady=10)

# Start the GUI main loop
window.mainloop()
```

Optimisation.py:

```
#
# The Optimisation methods are used to calculate the turns profile when spiral coils are stacked together.
#

import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt
from ResidualB import B0

pi = 3.14159265358979323846264338327795
mu0 = 4.0 * pi * 1.0e-7  # vacuum magnetic permeability

def TurnsOptimisation(tolerance, L0, L, d0, delta, h, R, a, w, rho, I, N, Method):
    M = int(L/w)  # number of the spiral coils; must be odd
    M = M + 1 if M % 2 == 0 else M + 2
    zm = np.array(list(map(lambda m: -((M - 1)/2) * w + w * m, range(M))))  # coordinates of the spirals for [-L/2,L/2]

    Q = int(L0/w)  # number of the induction sampling points; must be odd
    Q = Q + 1 if Q % 2 == 0 else Q + 2
    zq = np.array(list(map(lambda q: -((Q - 1)/2) * w + w * q, range(Q))))  # induction sampling points for [-L0/2,L0/2]
    b0 = np.array(list(map(lambda q: B0(zq[q]), range(Q))))  # sampling of the residual induction in Tesla

    plt.plot(zq, b0)
    plt.xlabel("z0, m")
    plt.ylabel("B0, T")
    plt.title("Residual induction")
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    # Some initial constants used in the functions
    d = d0 + 2.0 * delta  # total conductor diameter including the isolation
    R = d * int(R / d) + d / 2.0  # adjusted internal radius
    gamma = d / (2.0 * pi)
    theta1 = (2.0 * pi * R) / d - pi
    gtd1 = 2.0 * gamma * theta1 + d

    # Function in Eq. (32) in the report
    def Bz(x, q):
        Bz_value = 0.0
        for m in range(M):
            theta2 = theta1 + x[m] * np.tanh(a * x[m])
            gtd2 = 2.0 * gamma * theta2 + d
            sqrt_term = lambda gtd: (gtd ** 2 + 4.0 * (zq[q] - zm[m]) ** 2) ** 0.5
            value1 = gtd1 + sqrt_term(gtd1)
            value2 = gtd2 + sqrt_term(gtd2)
            value = np.log(value2 / value1) + gtd1 / sqrt_term(gtd1) - gtd2 / sqrt_term(gtd2)
            Bz_value = Bz_value + np.tanh(a * x[m]) * value
        return mu0 * I * Bz_value / (4.0 * pi * gamma)

    # Induction function used for testing the optimisation (step signature function)
    def Bz_test(x, q):
        Bz_value = 0.0
        for m in range(M):
            if x[m] < 0.0:
                theta2 = theta1 - x[m]
            else:
                theta2 = theta1 + x[m]
            gtd2 = 2.0 * gamma * theta2 + d
            sqrt_term = lambda gtd: (gtd ** 2 + 4.0 * (zq[q] - zm[m]) ** 2) ** 0.5
            value1 = gtd1 + sqrt_term(gtd1)
            value2 = gtd2 + sqrt_term(gtd2)
            value = np.log(value2 / value1) + gtd1 / sqrt_term(gtd1) - gtd2 / sqrt_term(gtd2)
```

```
            if x[m] < 0:
                Bz_value = Bz_value - value
            else:
                Bz_value = Bz_value + value
        return mu0 * I * Bz_value / (4.0 * pi * gamma)

    # Least square function in Eq. (31) used for the optimisation
    def fun(x):
        return sum((Bz(x, q) + b0[q])**2 for q in range(Q))

    # Derivative in Eqs. (33), (34) calculated using SymPy library (https://www.sympy.org/en/index.html)
    # During the calculations it may overflow, but will continue. Choose a proper "a" parameter.
    def dBz(x, m, q):
        b = 0.25 * I * mu0 * (a * ((0.5 * d + gamma * (theta1 + x * np.tanh(a * x)))**2 + (zm[m] -
zq[q])**2)**2.5 * (d
        + 2.0 * gamma * (theta1 + x * np.tanh(a * x)) + 2.0 * ((0.5 * d + gamma * (theta1 +
        x * np.tanh(a * x)))**2 + (zm[m] - zq[q])**2)**0.5) * (0.5 * gtd1 * ((0.5 * d + gamma * (theta1 +
        x * np.tanh(a * x)))**2 + (zm[m] - zq[q])**2)**0.5 - (0.5 * d + gamma * (theta1 +
        x * np.tanh(a * x))) * (0.25 * gtd1**2 + (zm[m] - zq[q])**2)**0.5 + (0.25 * gtd1**2 + (zm[m] -
        zq[q])**2)**0.5 * ((0.5 * d + gamma * (theta1 + x * np.tanh(a * x)))**2 + (zm[m] -
        zq[q])**2)**0.5 * np.log((d + 2.0 * gamma * (theta1 + x * np.tanh(a * x)) + 2.0 * ((0.5 * d +
        gamma * (theta1 + x * np.tanh(a * x)))**2 + (zm[m] - zq[q])**2)**0.5) / (gtd1 + 2.0 * (0.25 * gtd1**2
        + (zm[m] - zq[q])**2)**0.5))) / np.cosh(a * x)**2 + gamma * (0.25 * gtd1**2 + (zm[m] -
        zq[q])**2)**0.5 * (a * x / np.cosh(a * x)**2 + np.tanh(a * x)) * ((0.5 * d + gamma*(theta1 +
        x * np.tanh(a * x)))**2 + (zm[m] - zq[q])**2)**0.5 * (0.5 * (0.5 * d + gamma * (theta1 +
        x * np.tanh(a * x))) * (d + 2.0 * gamma * (theta1 + x * np.tanh(a * x)))*((0.5 * d + gamma * (theta1 +
        x * np.tanh(a * x)))**2 + (zm[m] - zq[q])**2) * (d + 2.0 * gamma * (theta1 + x * np.tanh(a * x)) +
        2.0 * ((0.5 * d + gamma * (theta1 + x * np.tanh(a * x)))**2 + (zm[m] - zq[q])**2)**0.5) +
        2.0 * ((0.5 * d + gamma * (theta1 + x * np.tanh(a * x)))**2 + (zm[m] - zq[q])**2)**2.0 * (0.5 * d +
        gamma * (theta1 + x * np.tanh(a * x)) + ((0.5 * d + gamma * (theta1 + x * np.tanh(a * x)))**2 + (zm[m]
        - zq[q])**2)**0.5) - ((0.5 * d + gamma * (theta1 + x * np.tanh(a * x)))**2 + (zm[m] -
        zq[q])**2)**2.0 * (d + 2.0 * gamma * (theta1 + x * np.tanh(a * x)) + 2.0 * ((0.5 * d + gamma * (theta1
        + x * np.tanh(a * x)))**2 + (zm[m] - zq[q])**2)**0.5)) * np.tanh(a * x)) / (gamma * pi * (0.25 *
gtd1**2 +
        (zm[m] - zq[q])**2)**0.5 * ((0.5 * d + gamma * (theta1 + x * np.tanh(a * x)))**2 + (zm[m] -
        zq[q])**2)**3.0 * (d + 2.0 * gamma * (theta1 + x * np.tanh(a * x)) + 2.0 * ((0.5 * d + gamma * (theta1
        + x * np.tanh(a * x)))**2 + (zm[m] - zq[q])**2)**0.5))
        return b

    # Calculation of the Jacobian in Eq. (33)
    def jac(x):
        J = np.zeros(M)
        for m in range(M):
            J[m] = 2.0 * sum((Bz(x, q) + b0[q]) * dBz(x[m], m, q) for q in range(Q))
        return np.array(J)

    # Calculation of the Hessian matrix in Eq. (34)
    def hess(x):
        HM = []
        for m in range(M):
            row = []
            for n in range(M):
                value = 2.0 * sum(dBz(x[m], m, q) * dBz(x[n], n, q) for q in range(Q))
                row.append(value)
            HM.append(row)
        HM = np.array(HM)
        return HM

    # Minimisation algorithm:
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#id1
    # We will use only these four methods: trust-exact (1), trust-krylov (2), trust-ncg (3), Newton-CG (4)
    x0 = np.full(M, N * 2.0 * pi)  # array with the initial degrees (rads), where N is the initial number of
turns
    if Method == 1:
        optimisation = 'trust-exact'
    elif Method == 2:
        optimisation = 'trust-krylov'
```

```
    elif Method == 3:
        optimisation = 'trust-ncg'
    else:
        optimisation = 'Newton-CG'

    result = minimize(fun, x0, method = optimisation, jac = jac, hess = hess, tol = tolerance)
    xs = np.array(result.x)  # array of the optimised values of the variable "hi" in Eq. (29)
    turns = (xs / (2.0 * pi)).astype(int)  # angles converted to the full turns
    xs_adj = turns * 2.0 * pi  # adjusted 'hi' for the full turns

    # Quality of the optimisation
    error = [lambda q: (Bz_test(xs_adj, q) + b0[q]) for q in range(Q)]
    error = np.array([func(q) for q, func in enumerate(error)])

    def plot_and_save_data(x, y, title, xlabel, ylabel, filename):
        plt.plot(x, y)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.title(title)
        plt.grid(True)
        plt.tight_layout()
        plt.show()

        data = np.column_stack((x, y))
        np.savetxt(filename, data, delimiter=",")

    def plot_and_save_data_with_legend(x1, y1, x2, y2, title, xlabel, ylabel, filename):
        plt.plot(x1, y1, color='blue', label='B0')
        plt.plot(x2, y2, color='green', label='B0 + Bz')
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.title(title)
        plt.legend()
        plt.tight_layout()
        plt.grid(True)
        plt.show()

        data = np.column_stack((x1, y1, y2))
        np.savetxt(filename, data, delimiter=",")

    # Plot 1: Turns profile
    plot_and_save_data(zm, turns, "Turns profile", "z0, m", "Full turns", "turns_profile.csv")

    # Plot 2: Error profile
    plot_and_save_data(zq, error, "Error profile", "z0, m", "Error, T", "error_profile.csv")

    # Plot 3: Induction profile before and after optimization
    plot_and_save_data_with_legend(zq, b0, zq, error, "Induction profile before and after optimization", "z0,
m",
                                   "Induction, T", "induction_profile.csv")

    # Calculation of the individual resistances of the spiral coils and the total resistance of the stack
    wire_resistance = []
    strip_resistance = []
    spiral_length = []
    for m in range(M):
        if xs_adj[m] < 0.0:
            theta2 = theta1 - xs_adj[m]
        else:
            theta2 = theta1 + xs_adj[m]
        gtd2 = 2.0 * gamma * theta2 + d
        # Spiral length and resistance
        val1 = (4.0 * gamma**2 + gtd1**2)**0.5
        val2 = (4.0 * gamma**2 + gtd2**2)**0.5
        length = ((gtd2/(8.0 * gamma)) * val2 - (gtd1/(8.0 * gamma)) * val1 +
                  (gamma / 2) * np.log((gtd2 + val2) / (gtd1 + val1)))
        spiral_length.append(length)
```

```
194.         wire_resistance.append(rho * length / ((pi / 4.0) * d0**2))
195.         strip_resistance.append(rho * length / (d0 * h))
196.
197.     # Writing the wire and microstrip resistances to the file
198.     data = np.column_stack((zm, spiral_length, wire_resistance, strip_resistance))
199.     csv_file = "length_resistance_profiles.csv"
200.     np.savetxt(csv_file, data, delimiter=",", header="zm(m), Length(m), R_wires(Ohms), R_strips(Ohms)",
comments='')
201.
202.     # Coil stack parameters
203.     total_spiral_length = np.sum(spiral_length)  # length (m)
204.     total_wire_resistance = np.sum(wire_resistance)  # resistance (Ohms)
205.     total_strip_resistance = np.sum(strip_resistance)  # resistance (Ohms)
206.     wire_current_power = total_wire_resistance * I**2  # power dissipated in the whole wire stack
207.     strip_current_power = total_strip_resistance * I**2  # power dissipated in the whole strip stack
208.
209.     with open('design_parameters_for_turns.txt', 'w') as file:
210.         file.write(f'total current through all spiral coils = {I} A\n')
211.         file.write(f'strip thickness = {h} m\n')
212.         file.write(f'conductor diameter/width without isolation/gap = {d0} m\n')
213.         file.write(f'total conductor diameter/width = {d} m\n')
214.         file.write(f'total substrate width, including the wire diameter = {w} m\n')
215.         file.write(f'residual induction length = {L0} m\n')
216.         file.write(f'total optimisation length = {L} m\n')
217.         file.write(f'adjusted internal radius = {R} m\n')
218.         file.write(f'total length of the spirals = {total_spiral_length} m\n')
219.         file.write(f'total wire resistance = {total_wire_resistance} Ohms\n')
220.         file.write(f'total strip resistance = {total_strip_resistance} Ohms\n')
221.         file.write(f'power dissipated in the whole wire stack = {wire_current_power} W\n')
222.         file.write(f'power dissipated in the whole strip stack = {strip_current_power} W\n')
```

ResidualB.py:

```
 1. #
 2. # Single-axis residual induction (T, Tesla) as a function of the z0 coordinate.
 3. # You can use a polynomial or any other fitting function of a single variable to describe the residual field.
 4. #
 5.
 6. # A residual field provided by the mmWave lab at the University of St. Andrews, Scotland.
 7. def B0(z0):
 8.     value = -5.859e-6 + 4.766114e-3 * z0 - 0.486506371 * z0**2 - 1.4609783504e1 * z0**3 + 4.2600403748e2 *
z0**4 +\
 9.             3.04437e4 * z0**5 - 8.75637e5 * z0**6 - 7.65903e6 * z0**7 + 1.46997e8 * z0**8
10.     return value
```