UROP Summer 2023 by Alan Li and George Smith

Introduction:

The accurate measurement of the electron Electric Dipole Moment (eEDM) stands as a pivotal task. An integral challenge in this pursuit has been the magnetic noise detected by the Quspins housed inside the magnetic shielding. While this noise impedes our ability to achieve a smaller systematic error, it also offers an exciting opportunity to innovate and implement novel techniques for noise reduction. Our initial approach leveraging Independent Component Analysis (ICA) yielded suboptimal signal decomposition outcomes. Subsequent investigations employing machine learning methodologies for blind source separation further elucidated the inherent complexities and challenges of the task. Given these outcomes, our focus shifted to dynamic compensation techniques, presenting a promising pathway for noise reduction. This guide will detail our systematic exploration and findings in dynamic compensation

What to do before you start reading the rest of this guide:

Read: The design of the n2EDM experiment https://doi.org/10.1140/epjc/s10052-021-09298-z

Dynamic stabilization of the magnetic field surrounding the neutron electric dipole moment spectrometer at the Paul Scherrer Institute https://doi.org/10.1063/1.4894158

Feedback for physicists: A tutorial essay on control, John Bechhoefer (First 7 Pages)

Make sure that you understand:

Singular Value Decomposition

The Moore-Penrose Pseudoinverse

Tikhonov regularization and condition number

Control theory, especially PID control & feedback

The important files that you will need are in OneNote at UltracoldEDM -> UROP2023 -> Important Stuff

Experimental Setup

A list of equipments we used for the dynamic compensation system:

One Bartington Mag-03 Magnetometer

One lab built bipolar amplifier (BOP)

One NI DAQ board

One Desktop(Ybfultracold)

Several cables

One oscilloscpe

One signal generator

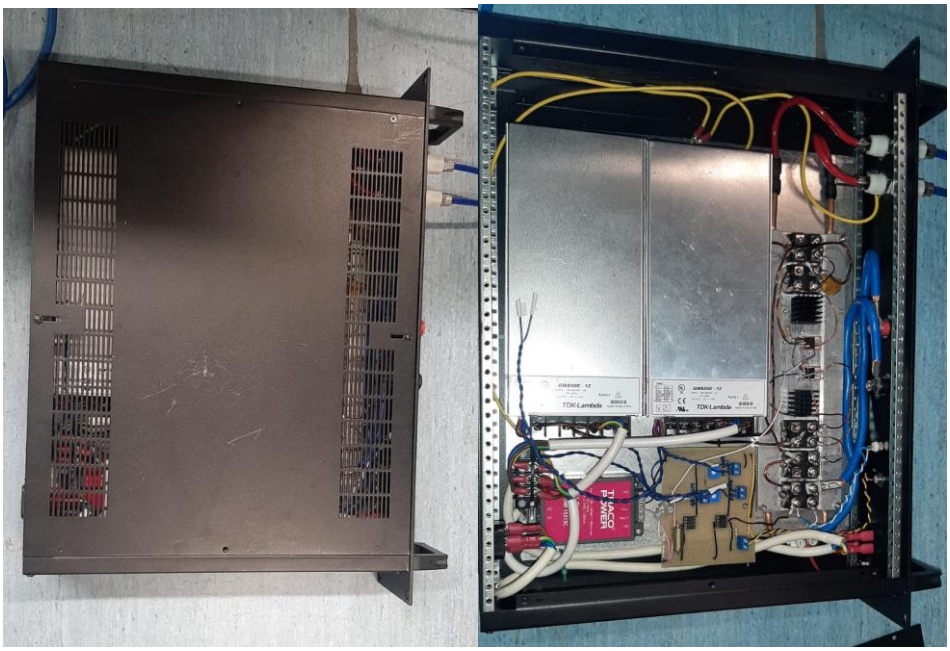


Fig.1 The lab built Bop

Fig.2 The bipolar amplifier is water cooled and is controlled by the switch on the right, flat is off

Fig.3 We added a resistor on the wire connecting from the bipolar amplifier output to the coil to measure the current through the coil. Note that you need two BNCs and you should measure their voltage difference in an oscilloscope to read the correct voltage, otherwise it is not grounded properly

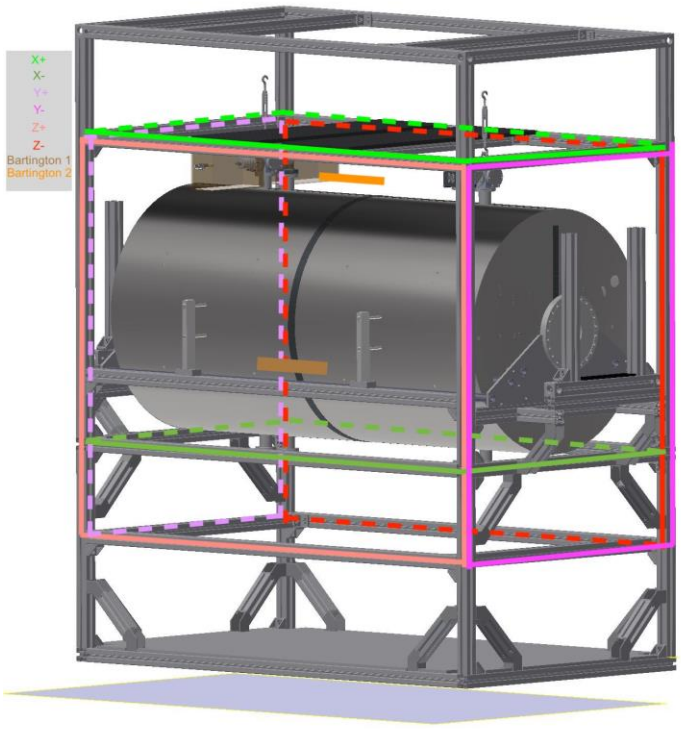


Fig.2 The Uedm Mu-metal shield and compensation coils

- The Bipolar Amplifier is connected to the analog output of the DAQ Board
- The Bartington is connected to the analog input of the DAQ Board
- We used Bartington 1 for our project
- We used the signal generator as a clock which is connected to the DAQ Board

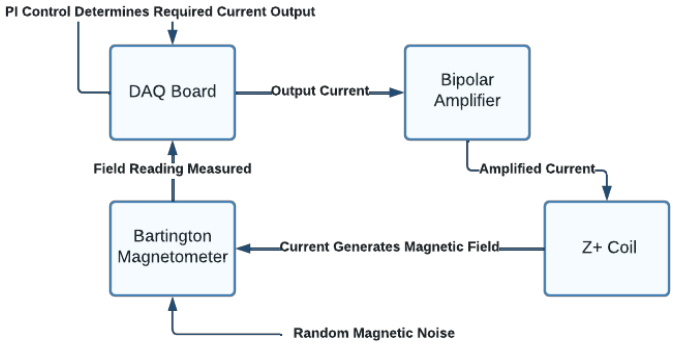


Fig.3 The dynamic compensation process flow chart

List of softwares:

NIMAX: we used it to check if the if the clock(signal generator) is working properly

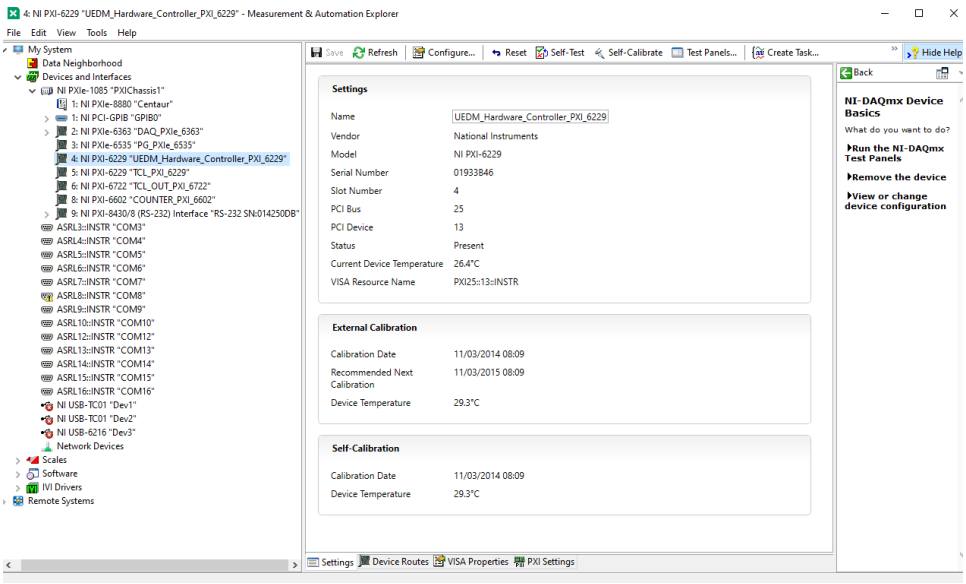


Fig. NIMAX interface, select the board you are connecting your clock to and press “Test Panels”

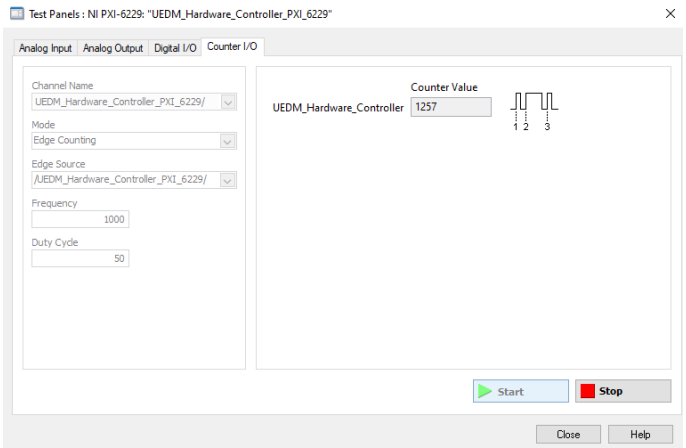


Fig. NIMAX Test Panels, go to “Count I/O” and change the mode to “Edge Counting” and select the channel that you connected your clock to, then press “start”

QZFM: This is used to start the QuSpins

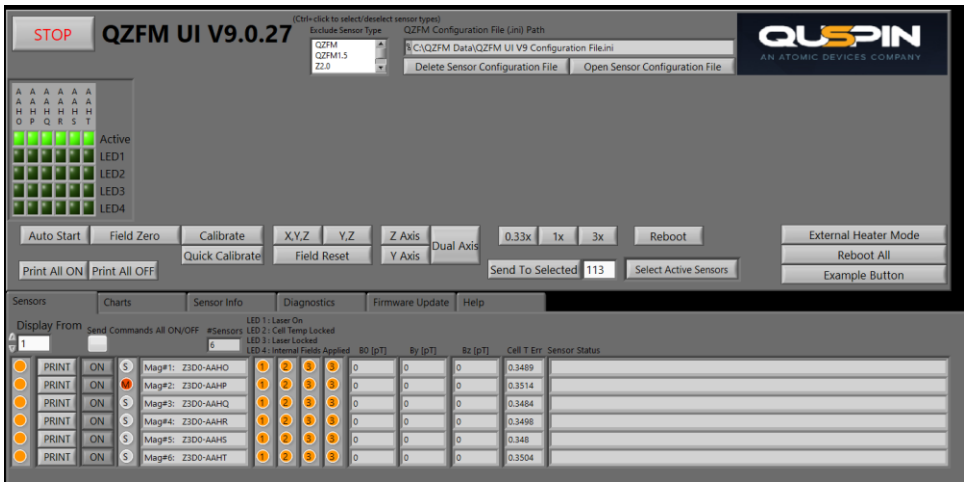


Fig. The QZFM interface. To start the Quspins, turn on all the switches first, then open the software and press “Auto Start” and wait until heating is complete, then press “X, Y, Z”, “Dual Axis” and we set our gain to “3X”. Then press “Field Zero” and wait until B field measurements are constant, after which press “Calibrate” which alters temperature slightly to test stability, all values should be close to 1 pF or under. If you wish to see the fields, you can click “PRINT ALL ON” to view the signals.

Signal Express: we used this software to take data for both the QuSpins and the Bartington

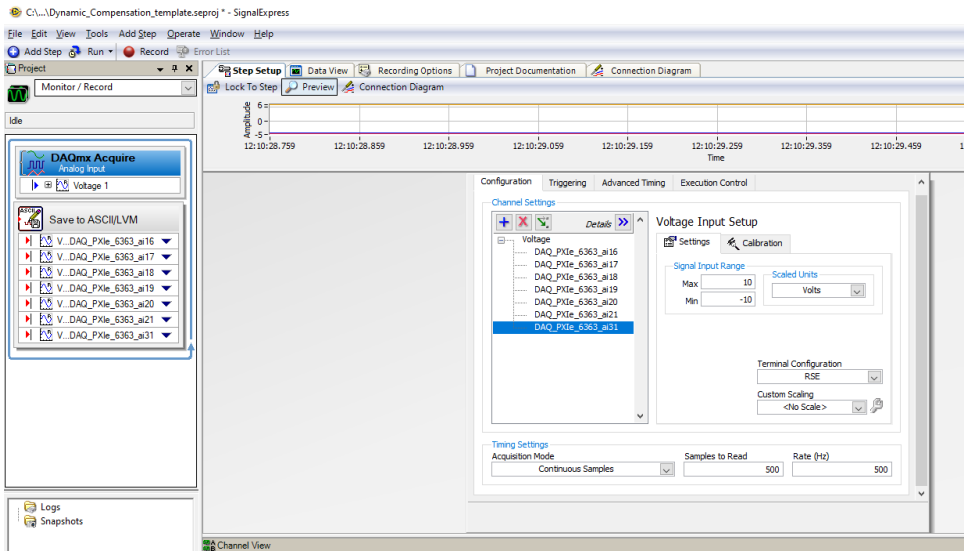


Fig. The Signal Express interface. To acquire data, hover yout cursor to the box on the left and right click, then go to Acquire Signals -> DAQmx Acquire -> Analog Input -> Voltage. After that you can add channels

by pressing the blue “+” button and choose the channel that you wish to add. You can also adjust the sampling rate in “Timing Settings”. To save signals, right click and the same location and go to Load/Save Signals -> Analog Signals -> Save to ASCII/LVM. After that you can go to “File settings” on the right and change the export file type to “Generic ASCII (.txt)” and you should also change the time axis preference to “Relative Time”. We take data for 10 minutes by clicking the arrow next to “Run” and clicked on “Configure Run” and ticked “For 600 Seconds” before clicking “Run”. Your data will save automatically to the file path you specified one it’s done.

VisionStudio 2019: This is the software we used to launch and edit the Magnetic Field Lock software.

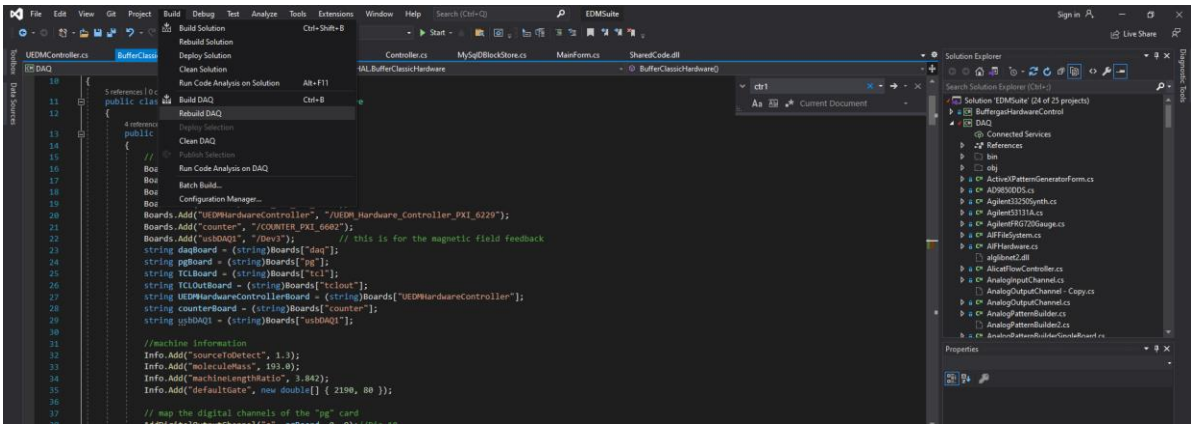


Fig. Vision Studio interface, open the software and open EDMSuite.Sln, then hover on “Build” and press “Configuration Manager”. After you have configured the projects, press “Clean Solution” then press “Build Solution” then press the “Start” Button on the menu bar to launch.

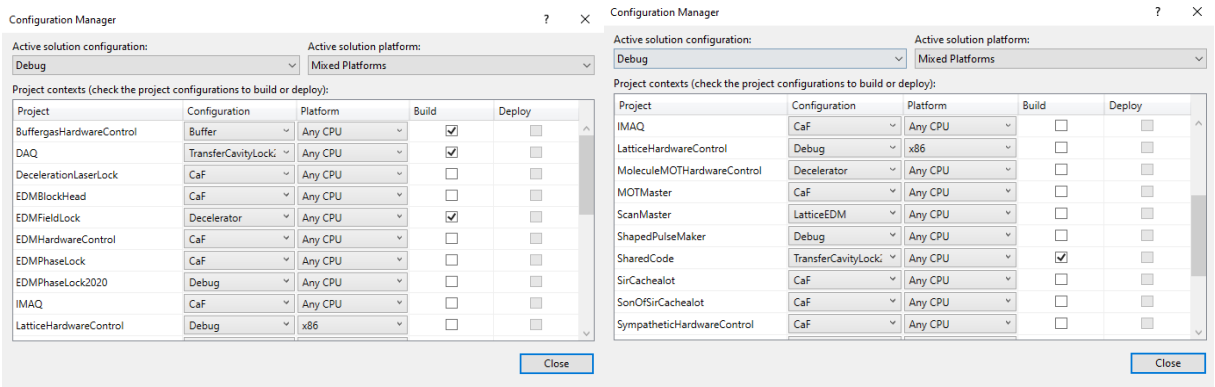


Fig. The Configuration Manager, Make sure that you tick the correct projects to build.

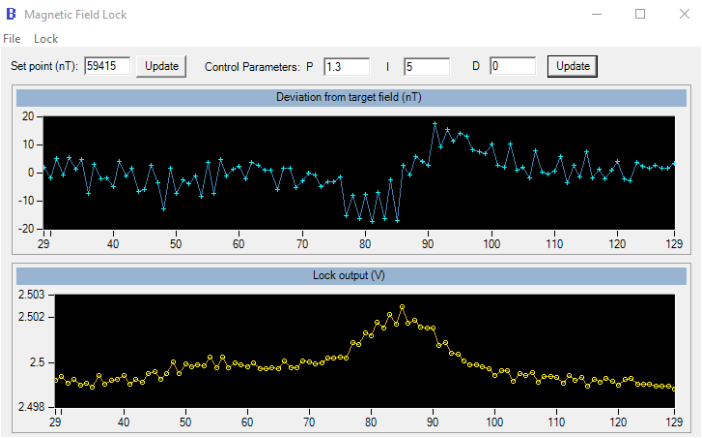


Fig. The Magnetic Field Lock interface

Methodology

Theory

Each fluxgate detector(j in total) has a linear response to current changes in each of the k coils. Proportionally factor matrix defined as:

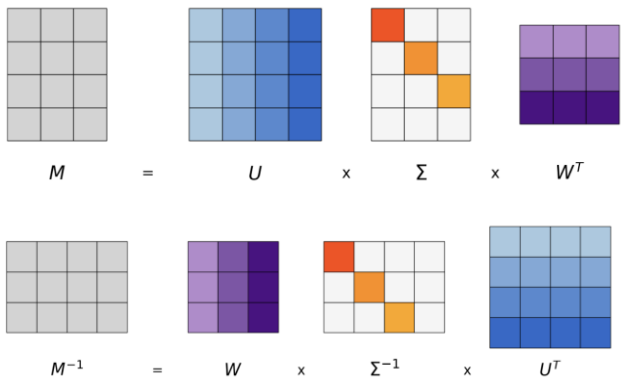
$$B_k = \sum_j M_{kj} I_j$$

We find the factors by passing different currents in one coil and measure how the output voltage of each detector changes. Repeat that for all of the coils and find the corresponding gradients of best fit lines gives you the PF matrix.

The compensation current is calculated by inverting the PF matrix

$$\Delta I_j = \sum_k M_{jk}^{-1} \cdot (B_k^{target} - B_k^{read})$$

You may notice that the PF matrix may not be a square matrix, hence we will need the Moore-Penrose pseudoinverse:



Data Taking

Try to send different currents to one of the coils and measure the output voltage response of one of the detectors. Find the gradient of the best fit line, and that will be one matrix element of your proportionality matrix. After making the correct scaling , and repeating over all coil-detector combinations, you will get the proportionality matrix with units nT/A.

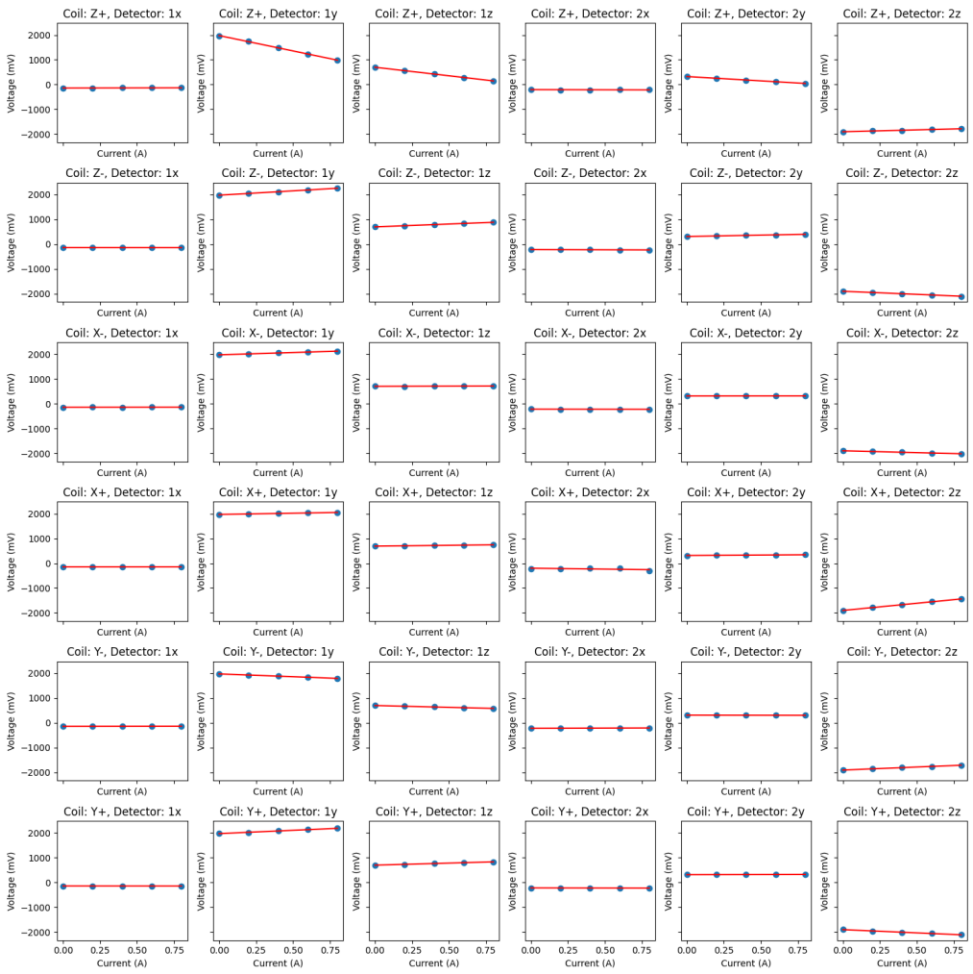


Fig.4 The PF Matrix, here we have 6 coils and 6 detectors(each Bartington measures 3 axes)

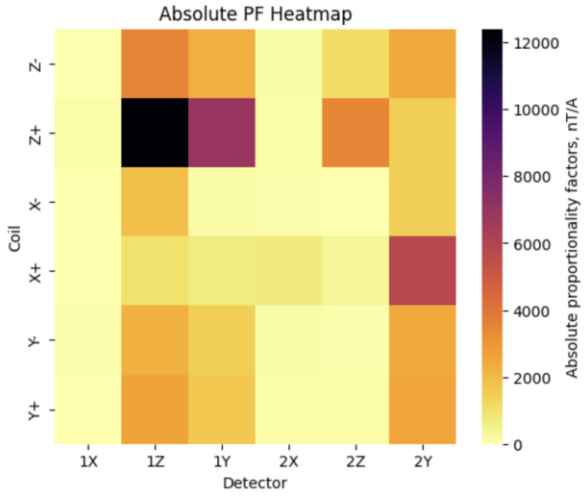


Fig.5 The absolute PF Matrix Heatmap

The need for regularisation for the inverse problem.

Condition Number – this is a measure of the sensitivity to changes in the unput and their effect on the output. A low condition number means that small fluctuations in the currents leads to small fluctuations in the field calculated while a high condition number means small changes lead to large changes. (This can be seen as a mapping that is no longer smooth, and therefore can create undesirable jumps in the B_k calculated. (Note that when we say fluctuations here, we mean changes and variations in the currents not due to our changing of them).

Condition number is important for us, as M is likely to be ill-conditioned (high condition number) due to a large ratio between its largest and smallest elements. The condition number can be formally given by:

The ratio of the largest to smallest singular values: $\kappa(M) = \frac{\sigma 1}{\sigma_n}$

(σ_i are the singular values of M, where they are ordered monotonically decreasing $\sigma_1 > \sigma_2 > \ldots > \sigma_n$)

Also: $\kappa(M) = \|M\|\left\|M^{\dagger}\right\|$ (Where M-dagger is the Moore-Penrose Pseudoinverse).

An infinite condition number means that the matrix is non-invertible. A high condition number means that inversion is computationally difficult.

This is easiest to see in terms of the singular values. If there is a small singular value, M will be close to singular and therefore difficult to invert. From the SVD of M, consider the elements $(\Sigma)_{jj} = \sigma_j$, then the elements of Σ^{-1}, which constitutes M^{-1}, are given by $1/\sigma_j$. and hence will blow up as $\sigma_i \to 0$. A computer will not like this.

Tikhonov Regularisation / Ridge Regression.

For ill-conditioned problems, regularisation can be used to improve numerical stability and computation.

The general matrix inversion problem can be cast as:

$$min\ \|M\boldsymbol{I} - \boldsymbol{B}\|_2^2$$

(Minimising the Euclidean norm squared, just a general least-squares problem)

We can modify this by adding an additional term to the minimisation.

$$min\ \|M\boldsymbol{I} - \boldsymbol{B}\|_2^2 + \|\Gamma\boldsymbol{I}\|_2^2$$

Γ is a Tikhonov matrix which is often chosen to be a scalar multiple of the Identity matrix (We use $\Gamma = \beta I$.

The SVD now results in $\Sigma_{jj}^{-1} = \frac{1}{\sigma_j} \longrightarrow \frac{\sigma_j}{\sigma_j^2+\beta^2}$.

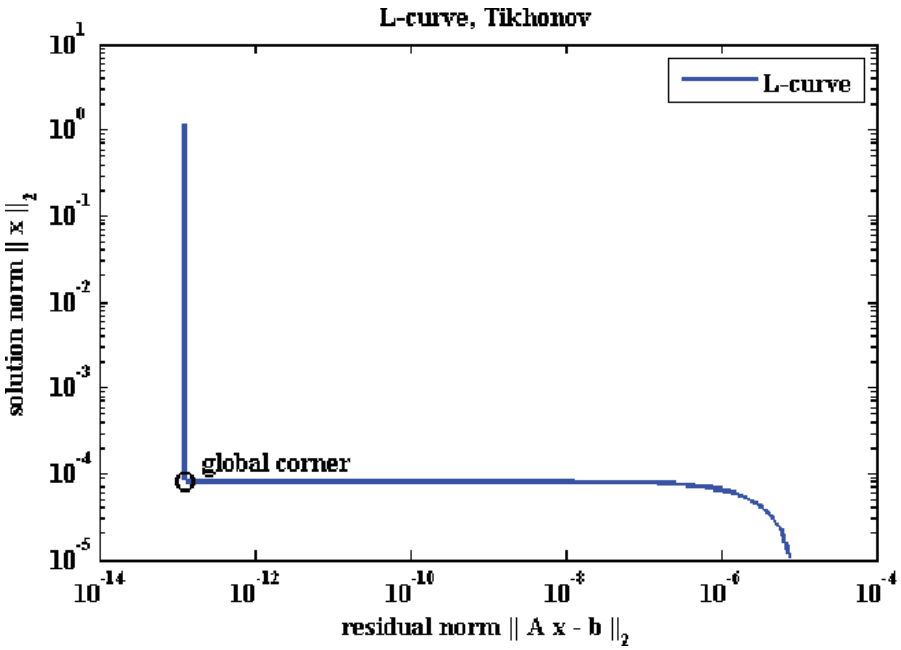
This will result in a less accurate pseudoinverse. (i.e M x M-dagger will be further away from the identity matrix, and so our calculated current to account for a given deviation in the magnetic field will be less accurate and cancelling it out. A suitable regularisation parameter, β, is required to compromise between the two terms in the modified minimisation problem. This can be tested experimentally, and seeing what value of β gives the best results.

We replace β with 10^r, as r is on a nicer scale to work with (see later)

We made a simulation to see if we could predict what would be a good parameter choice.

One common method, though not one we used (but is of a similar style) is the L-curve method.

This involves a log-log plot of the two terms in the modified equation ()for different values of r and taking a value that corresponds to the "corner" of the "L" shape.



This aims to minimise them both simultaneously.

Our simulation uses proxy cost functions for these two competing factors.

First, we simulate a set of 30 (variable) random magnetic fields, $\mathbf{B}^{rand}$ (normally distributed about the same values that we measure in the lab). For a range of values of r, we use our M and find the regularised pseudoinverse. We use this to find our vector of correction currents, $\mathbf{I}^{sim}$.

For a metric of the current term in (), we use the rms of $\mathbf{I}^{Sim.}$

$$\Gamma = \sqrt{\frac{1}{6}\sum_{j=1}^{6}\left(I_k^{Sim}(r)\right)^2}$$

(6 is used as that is the number of coils)

We want to minimise Γ.

For a metric for the first term, we introduce:

$$B_k^* = B_k^{rand} + \sum_j M_{kj} \cdot I_j^{sim}(r)$$

This is a measure of how well the inverse works,

$$b = \sqrt{\frac{1}{K}\sum_{k=1}^{K}\left(B_k^{rand}\right)^2}$$

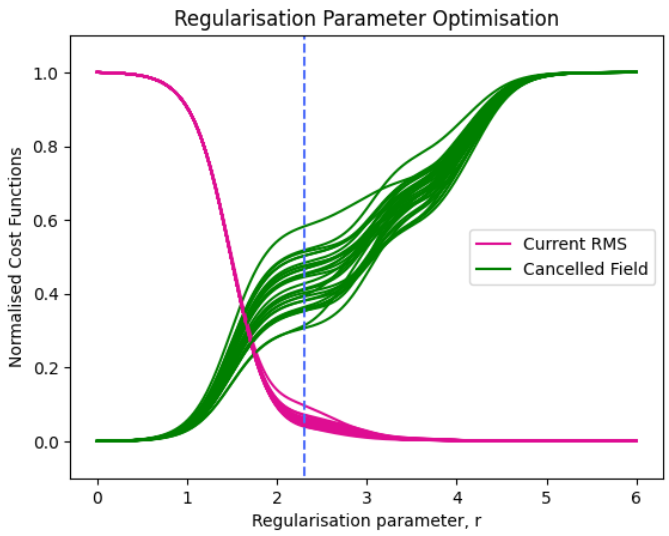
$$b^* = \sqrt{\frac{1}{K}\sum_{k=1}^{K}\left(B_k^{*}\right)^2}$$

$$R = \frac{b^*}{b}$$

Where R = 0 means perfect cancellation.

We normalised both Γ and R to the unit interval using min-max normalistion (just a linear scaling)

Plotting these for each value of B:



There is one line of each colour for each set of random magnetic field values (n = 30 here).

We wanted to minimise these simultaneously. We did this by: For each iteration, n, we summed the normalised Γ and R for each value of r (Has the same effect as averaging over the value for the nth set). We took the value of r corresponding to the minimum of these sums. This gave us r = 2.036.

Taking a greater value for n gave the same mean, and the graphs were similar, but just larger "bulges" where the values R and Γ spread.

Ultimately, we can only verify the results and predictions from the simulations with experiment. Once fully running (with 6 coils and more magnetometers etc). It would be best to try running the PI loop with matrix elements found with different regularisation parameters, and see which has the best improvements. This will allow us to see if regularisation is actually needed (which we anticipate should be, but can not verify this yet) and how to find the best parameter.

Control Theory and PI Loops

Reading “Feedback for Physicists” will give the main points to understand.

Ultimately, we want to be able to use the magnetic field measurements, use our matrix inversion to find a current required for a certain magnetic field, and to send a determined current to cancel out these magnetic fields. We want to be able to run this continuously while the experiment would be running, in order to reduce the magnetic noise during the experiment.

Comparing our experiment to a general feedback loop.

- r(t) - Target for our controlled quantity **(B^{Target})**
- y(t)- Measured output value of this quantity - **B^{Read}**
- K(t)- Control Law (Coils outputting a certain current)
- G(t)- System (Our magnetometer array)
- e(t) - Error given by r(t) - y(t) – equivalent to **ΔI** or **ΔB** (With matrix inversion, these become interchangeable)
- u(t) - Response to control law (See below)

<u>PID Controller</u>

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)\ d\tau + K_d \cdot \frac{de(t)}{dt}$$

Our control law determines what signal we need to output to correct for the error.

There are 3 parts to a PID controller: proportional, integral, derivative (hence PID). These act in different ways to reduce the error. “Proportional” is used to directly correct for the error by linearly altering the output reading so that it will match the target and the error will vanish. However, eventually the error will become too small, and the error will fail to reduce any further; this is why the “Integral” is needed. As the error, and hence proportional term, become small and stationary, the integral term will continue to grow to account for these residual errors. This will be sufficient to reduce the error to effectively 0. The derivative term acts differently, as this is uses the rate of change of the error to anticipate how it will change, and can therefore act “early” to reduce this.

K_p, K_i, and K_d are tuning parameters which are selected to give the best response. These will depend on the system, and are manually changeable.

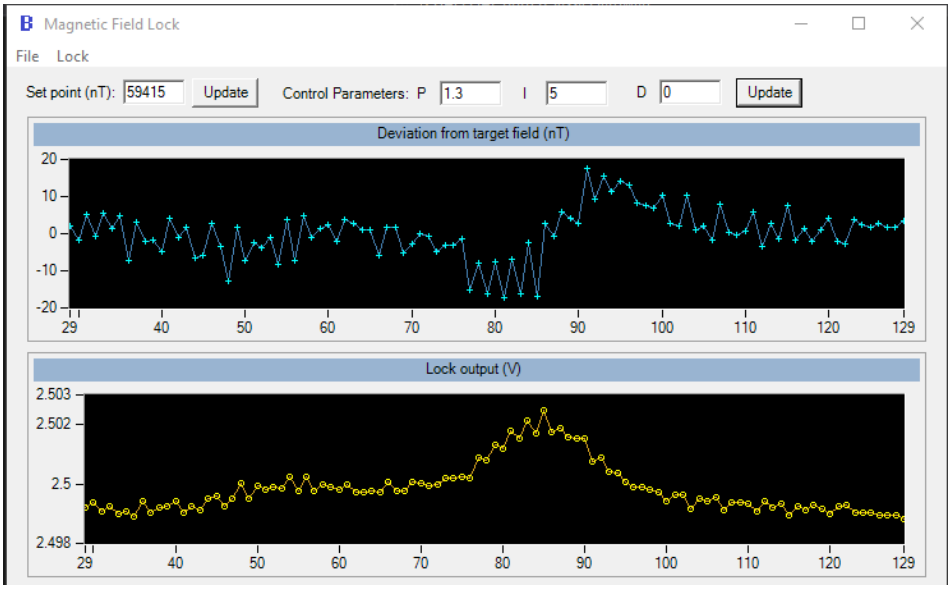
$$I_j^n = K_j^P \cdot \Delta I_j^n + K_j^I \cdot \sum_{t=0}^{n} \Delta I_j^t$$

This is the PI control law applied to our experiment. This updates the current elements I_j for the nth iteration of the loop. Each coil will have its own PI parameters. There is no derivative term here as it is not useful when dealing with noise. The derivative is used to predict future trends, but the derivative is not representative of the future of the error as the derivative of the noise is often not smooth. The integral is now a sum, as our data is discretely sampled. **ΔI** is our error here, which is equivalent to the measured **ΔB**.

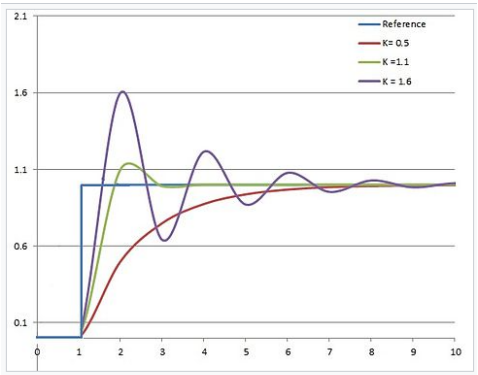
Choosing PI Parameters

We determined these manually, though there are methods which should also provide suitable values. We did not investigate these methods as we wanted to investigate and understand physically the impact of different parameter values. When more coils are implemented, these may be necessary.

For manual selection, we needed to see what values would give the "best" response. The FieldLock Program creates an app where the PID parameters can be altered easily, and we can see the



deviation of the magnetic field, as well as the voltage output from the DAQ2 board to the coil.(P,I,D are K_P, K_I, and K_D, respectively.



Response of PV to step change of SP vs time, for three values of K_p (K_i and K_d held constant)

Firstly, it is best to set I to 0 (D will always be 0 for us) and alter this until we see a suitable response which returns to a low deviation quickly, without oscillating about the setpoint. We changed the target on the program by several hundred nT and looked at the response.

We found P = 1.3 to be a good value, though there will likely be a slight improvement in the range 1.2 – 1.5

We then added an I term, again aiming to reduce the oscillations after changing the target. The deviation graph looked like classical critical dampening, which is ideal. We found 5 to be a good value for I. Again, there was a fair range of suitable values. It may be best to take data while using different values of I to see if a particular value is significantly better, or if there is any real importance within a certain range.

One important indicator to see if the cancellation was working in real time was to look at the two graphs on the program. If the cancellation is poor, they will almost mirror each other perfectly, whereas for good cancellation, the voltage graph will vary but the deviation graph will be a lot more stable around 0. Another test to see the effect of cancellation is to place a metal object, like a phone, next to the Bartington and see how well the system reacts.

Methods like the “Ziegler–Nichols method” give a more automated approach to find tuning parameters based on properties of the response like the oscillation period and decay time of the output readings.

Each coil will need its own tuning parameters, something that we have not investigated. We would expect that each coil will have similar, or in an idealised case identical, P and I values. We have tested finding parameters with one coil and one sensor, so ideally this could be repeated for each coil and that will be sufficient. We are unsure however if it may become harder if we have multiple coils running at the same time, and if they do work cohesively as they should do. It may be difficult to see if the parameters are good, as the effect may be more pronounced for certain sensors more than others. Ultimately, trying each coil separately should be the best way to start this.

EDMFieldLock (Mainform.cs)

This is the c# file for running the PID program on the ultracold machine. It needs to be built with “SharedCode” and “BufferClassicHardware”.

Important parts of code for running and editing

```
// constants
const double FIELD_PER_VOLT_INPUT = 10000;      // in units of nT/V
const int SAMPLE_CLOCK_RATE = 50;
const int SAMPLE_MULTI_READ = 25;               // this is how many samples to read at a time, sets a limi
const int GUI_UPDATE_EVERY = 1;                 // if you multiply this by SAMPLE_MULTI_READ and divide by
const int LOCK_UPDATE_EVERY = 1;                // this is how often the lock is updated in terms of SAMPL
const double OUTPUT_LIMIT_LO = 0;               // (V). this sets the output limit.
const double OUTPUT_LIMIT_HI = 5;               // (V). this sets the output limit.
const double OUTPUT_ZERO = 2.5;                 // (V). this sets the zero output level.
const double INPUT_LOW = -10;                   // lower limit to Bartington input
const double INPUT_HIGH = 10;                   // upper limit to Bartington input
const int OUTPUT_RAMP_STEPS = 50;
const int OUTPUT_RAMP_DELAY = 100;
const int CURRENT_SETTLE_TIME = 1000;

double setPoint = 0.0;                          // in volts
double lockOutput = OUTPUT_ZERO;                // in volts
double proportionalGain = 1.0;
double integralGain = 0.0;
double derivativeGain = 0.0;
double lastDeviation = 0.0;
double lastIntegral = 0.0;
double lastOutput = 0.0;
double M = 3500.0; // calibration from field reading to required current nT/A (1Y with coil Z- used here)
    //will later replace with matrix of all M_jk
double currentVoltage = 1.095; // (A/V) calibration from current coil to DAQ Output voltage
int bartingtonVoltageField = 10000; //(1V input = 10,000 nT)
```

FIELD_PER_VOLT_INPUT – The bartington magnetometers give a voltage output reading, which we calculate the magnetic field reading from with the known calibration 1V = 10,000 nT.

SAMPLE_CLOCK_RATE – This is the frequency at which the system will update

SAMPLE_MULTI_READ – This is the number of samples taken in one batch of data (i.e in the time given by 1/ SAMPLE_CLOCK_RATE) We take several samples in each time period and average over these. This will reduce the impact of high frequency noises, which will be averaged out here as it acts

on a much smaller timescale. This gives an effective maximum cutoff frequency of SAMPLE_CLOCK_RATE / SAMPLE_MULTI_READ.

GUI_UPDATE_EVERY -If you multiply this by SAMPLE_MULTI_READ and divide by SAMPLE_CLOCK_RATE you get the GUI update interval in seconds.

LOCK_UPDATE_EVERY -This is how often the lock is updated in terms of SAMPLE_MULTI_READs (same idea as GUI update interval above)

OUTPUT_LIMIT_LO = 0;

OUTPUT_LIMIT_HI = 5;

OUTPUT_ZERO = 2.5;

The above 3 constants determine the range of voltages to be output by the DAQ2 board. We have a 2.5V DC offset and can go between 0-5 V. These are limits, and if crossed the program will change the setpoint value in order to prevent problems with the hardware.

```
double setPoint = 0.0;                          // in volts
double lockOutput = OUTPUT_ZERO;                // in volts
double proportionalGain = 1.0;
double integralGain = 0.0;
double derivativeGain = 0.0;
double lastDeviation = 0.0;
double lastIntegral = 0.0;
double lastOutput = 0.0;
double M = 3500.0; // calibration from field reading to required current nT/A (1Y with coil Z- used here)
    //will later replace with matrix of all M_jk
double currentVoltage = 1.095; // (A/V) calibration from current coil to DAQ Output voltage
int bartingtonVoltageField = 10000; //(1V input = 10,000 nT)

// 1V input = bartingtonVoltageField * (1/M) * (1/currentVoltage)
```

setpoint –

lockOutput –

proportionalGain – P tuning parameter

integralGain – I tuning parameter

derivativeGain – D tuning parameter

-The 3 above parameters have methods to be manually changed in the GUI

lastDeviation – the error read, this will update in each iteration

lastOutput – the previous Voltage output, will update in each iteration.

M – This is our Matrix element. As we only have 1 coil and 1 sensor, it is just a scalar, and the inverse is simply the reciprocal.

currentVoltage – this is the calibration we find between the output Voltage from the DAQ board, and the current produced in the coils (This takes into account the affect of the gain on the BOP etc) The program works in terms of Voltage, rather than current which we have been using so far but they are effectively interchangeable terms.

```
private object lockParameterLockObject = new Object();
private void UpdateLock()
{
    lock(lockParameterLockObject)
    {
        double meanDeviation = 0.0;
        foreach (double j in lockFieldData) meanDeviation += 1/(M * currentVoltage) * j / lockFieldData.Count;
        //we don't need meanDeviation ->> we do!! need it, but still need our Matrix calibration
        //currentVoltage term needed to convert needed current to output voltage

        //double deltaCurrent = 0.0;
        //foreach (double j in lockFieldData) deltaCurrent += 1/M * j;
        // 1/M for the inverse transformation



        double p, i, d, dt;
        dt = (double)LOCK_UPDATE_EVERY * (double)SAMPLE_MULTI_READ / (double)SAMPLE_CLOCK_RATE; // Time between this lock update and the last
        p = - proportionalGain * meanDeviation;
        i = - integralGain * meanDeviation * dt + lastIntegral;
        //d = - derivativeGain * (deltaCurrent - lastDeviation) / dt;
        //lockOutput = p + i + d + OUTPUT_ZERO;
        lockOutput = p + i + OUTPUT_ZERO; // d removed for PI loop
```

This method uses the PID controller to update the voltage output required.

meanDeviation uses the average of the elements in the group created from SAMPLE_MULTI_READ and then uses our proportionality factor and other calibrations (The calibration of the Bartington Voltage reading to the magnetic field is already used elsewhere in the code).

We have commented out d as it wasn't used, but keeping it at 0 will have the same effect.

This is all equivalent to

$$I_j^n = K_j^P \cdot \Delta I_j^n + K_j^I \cdot \sum_{t=0}^{n} \Delta I_j^t$$

The following methods are for configuring hardware and being able to read in and out of the correct channels on the DAQ board.

```
private void ConfigureAnalogOutput()
{
    if (!Environs.Debug)
    {
        analogOutputTask = new Task("field lock analog output");
        AnalogOutputChannel outputChannel = (AnalogOutputChannel)Environs.Hardware.AnalogOutputChannels["bFieldFeedbackOutput"];
        outputChannel.AddToTask(analogOutputTask, OUTPUT_LIMIT_LO, OUTPUT_LIMIT_HI);
        analogWriter = new AnalogSingleChannelWriter(analogOutputTask.Stream);
        analogWriter.WriteSingleSample(true, OUTPUT_ZERO); // start out with no current
    }
}
```

This searches the ClassicBufferHardware file for an input channel called "bfieldFeedbackOutput" and will use this to send out the required current/voltage.

```
private void ConfigureAnalogInput()
{
    if (!Environs.Debug)
    {
        analogInputTask = new Task("field lock analog input");
        AnalogInputChannel inputChannel = (AnalogInputChannel)Environs.Hardware.AnalogInputChannels["bFieldFeedbackInput"];
        CounterChannel clockChannel = ((CounterChannel)Environs.Hardware.CounterChannels["bFieldFeedbackClock"]);
        inputChannel.AddToTask(analogInputTask, INPUT_LOW, INPUT_HIGH);
        analogReader = new AnalogSingleChannelReader(analogInputTask.Stream)
        {
            SynchronizeCallbacks = true
        };
        analogInputTask.Timing.ConfigureSampleClock(
            clockChannel.PhysicalChannel,
            SAMPLE_CLOCK_RATE,
            SampleClockActiveEdge.Falling,
            SampleQuantityMode.ContinuousSamples
            );
```

This does the same, but for the inputs to the daq board. “bFieldFeedbackInput” will be used for the data read into the program. “bFieldFeedbackClock” is needed for a counter for the program. The clock on the computer isn’t accurate enough for this, so other hardware counters are used.

BufferClassicHardware

This contains methods to configure the inputs and outputs on the DAQ board.

```
//Magnetic feedback channels
AddAnalogInputChannel("bFieldFeedbackInput", usbDAQ1 + "/ai1", AITerminalConfiguration.Differential);
AddAnalogOutputChannel("bFieldFeedbackOutput", usbDAQ1 + "/ao1", 0, 5);
AddCounterChannel("bFieldFeedbackClock", usbDAQ1 + "/pfi0");
```

These are the necessary lines for the PID controller.

The first line adds an input channel called “bFieldFeedbackInput”, which it located on usbDAQ1 (hardware board) at position AI1 (Analog Input 1) (The final argument relates to the method of data acquisition). The other lines work similarly.

For increasing the number of coils and magnetometers, it is necessary to add more lines of code like these, an output channel for each coil, and an input channel for each magnetometer. Then the methods in EDMFieldLock will need modifying to include reading data from each input, and outputting a certain voltage for each coil. Each coil will have its own PID parameters. Ultimately, the program should read in *K* field readings, and each of the *J* PID loop will use these and output to the jth coil. Additionally, the proportionality factor matrix needs to be implemented into the code. Our algorithm for this is written in Python. It may be easier to just manually copy across the found M^{-1} into the c# file, or just replicate it (limited c# knowledge so unsure). The rest of the code should follow from the above, with lots of replication and indexing/loops to ensure each PID loop receives the “right” information i.e the correct matrix elements for each *meanDeviation* for each coil.

Explaining the rest of the code

The rest of the code revolves around taking the data and getting it to a state which can be used for the PID loop. The code also produces a GUI of the deviation of the field from the target, as well as the voltage output. I will skip over the blatantly obvious methods.

```
public void StartApplication()
{
    Application.Run(this);
}

private void StopApplication()
{
    if (running) StopAcquisition();
}

private void ClearGUI()
{
    deviationGraph.ClearData();
    outputGraph.ClearData();
}

private void UpdateGUI()
{
    deviationGraph.Plots[0].PlotYAppend((double[])meanDeviationData.ToArray(Type.GetType("System.Double")));
    deviationPlotData.Clear();
    meanDeviationData.Clear();

    outputGraph.Plots[0].PlotYAppend((double[])outputPlotData.ToArray(Type.GetType("System.Double")), LOCK_UPDATE_EVERY);
    outputPlotData.Clear();
```

UpdateGUI – updates the graphs on the GUI using the new values of the deviation and output voltage.

```
private void InitialiseSetPoint()
{
    if (!Environs.Debug)
    {
        double[] dataArray = analogReader.ReadMultiSample(SAMPLE_MULTI_READ);
        double dataMean = 0.0;
        for (int i = 0; i < dataArray.Length; i++) dataMean += dataArray[i] / dataArray.Length;
        setPoint = dataMean;
        setPointTextBox.Text = ((int)(setPoint * FIELD_PER_VOLT_INPUT)).ToString();
    }
```

This takes the reading from the Analog Input and takes the average. It creates a starting setpoint for the program based on the initial background reading. This value can be manually altered in the textbox on the GUI.

```
private void CounterCallBack(IAsyncResult result)
{
    // read the latest data from the analog input
    double[] data;

    if (!Environs.Debug)
    {
        data = analogReader.EndReadMultiSample(result);
    }
    else
    {
        Random r = new Random();
        data = new double[SAMPLE_MULTI_READ];
        for (int i = 0 ; i < SAMPLE_MULTI_READ ; i++)
        {
            data[i] = 2 * r.NextDouble() - 1;
        }
    }

    // start the counter reading again right away
    if (!Environs.Debug && running)
        analogReader.BeginReadMultiSample(
            SAMPLE_MULTI_READ,
            new AsyncCallback(CounterCallBack),
            null
            );
```

```
// start the counter reading again right away
if (!Environs.Debug && running)
    analogReader.BeginReadMultiSample(
        SAMPLE_MULTI_READ,
        new AsyncCallback(CounterCallBack),
        null
        );

// if reset, update set point
if (reset) ResetLockPoint(data);

// deal with the data
StoreData(data);

if (lockField && (sampleCounter % LOCK_UPDATE_EVERY == 0) && running) UpdateLock();

// update the gui ?
if (sampleCounter % GUI_UPDATE_EVERY == 0)
{
    UpdateGUI();
}
sampleCounter++;
```

This takes a set of data points corresponding to the value of SAMPLE_MULTI_READ. It then continues the counter until it gets to the next time at which it should resume sampling.

```
private void StoreData(double[] data)
{
    // create an array of the deviations of each point from the target
    double[] deviationArray = new double[data.Length];
    for (int i = 0; i < deviationArray.Length; i++) deviationArray[i] = data[i] - setPoint;
    lockFieldData.AddRange(deviationArray);

    // convert voltage into a field for plotting
    double[] fieldArray = new double[deviationArray.Length];
    for (int j = 0; j < deviationArray.Length; j++) fieldArray[j] = deviationArray[j] * FIELD_PER_VOLT_INPUT;
    deviationPlotData.AddRange(fieldArray);

    // add to averaged deviation array
    double mean = 0.0;
    for (int k = 0; k < fieldArray.Length; k++) mean += fieldArray[k] / fieldArray.Length;
    meanDeviationData.Add(mean);
}
```

These take the data and put into an array so that it can be used for the PID and plotted in the GUI. The 2nd block here uses our calibration from the Bartington voltage reading to the magnetic field it detects.

```
private void ResetLockPoint(double[] data)
{
    double mean = 0.0;
    for (int i = 0; i < data.Length; i++) mean += data[i] / data.Length;
    setPoint = mean;
    setPointTextBox.Text = ((int)(setPoint * FIELD_PER_VOLT_INPUT)).ToString();
    reset = false;
}
```

This is the same as the InitialiseSetPoint method, but can be implemented at any time when the program is running, i.e the user can write into the textbox on the GUI and update this value.

```
double p, i, d, dt;
dt = (double)LOCK_UPDATE_EVERY * (double)SAMPLE_MULTI_READ / (double)
p = - proportionalGain * meanDeviation;
i = - integralGain * meanDeviation * dt + lastIntegral;
//d = - derivativeGain * (deltaCurrent - lastDeviation) / dt;
//lockOutput = p + i + d + OUTPUT_ZERO;
lockOutput = p + i + OUTPUT_ZERO; // d removed for PI loop

// if reached voltage limit, reset lock point, otherwise update lock
if (lockOutput >= OUTPUT_LIMIT_HI || lockOutput <= OUTPUT_LIMIT_LO)
{
    reset = true;
```

After the PID part of the code, each time the voltage updates accordingly, the code checks that this won't exceed any voltage limitations put in place. If not, it continues as normal and repeats. If there is a problem, it will reset the setpoint to a value that won't results in the voltage leaving its bounds limits.

```
Windows Form Designer generated code
```

Under this tab, there is the code for producing the GUI. Ideally, this can be modified to include the deviation at each Bartington, and the voltage output to each coil.